RealView Debugger
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 User Guide

Preface
About this book ........................................................................................................ viii
Feedback .................................................................................................................. xv

Chapter 1 Starting to use RealView Debugger
1.1 Starting RealView Debugger ............................................................................. 1-2
1.2 Using RealView Connection Broker ............................................................... 1-6
1.3 RealView Debugger directories ...................................................................... 1-8

Chapter 2 Working with Images
2.1 Loading images ............................................................................................... 2-2
2.2 Managing images ............................................................................................ 2-8
2.3 Working with symbols ..................................................................................... 2-16
2.4 Working with multiple images ......................................................................... 2-17
2.5 Unloading and reloading images ..................................................................... 2-19

Chapter 3 Controlling Execution
3.1 Submitting commands ..................................................................................... 3-2
3.2 Defining execution context .............................................................................. 3-3
3.3 Using Execution controls ............................................................................... 3-6
3.4 Working with the Debug menu ....................................................................... 3-9
3.5 Controlling debugging ................................................................................... 3-13
Contents

Chapter 4  Working with Breakpoints
4.1  Breakpoints in RealView Debugger ....................................................... 4-2
4.2  Setting breakpoints quickly ................................................................. 4-5
4.3  Using simple breakpoints ................................................................... 4-10
4.4  Setting conditional breakpoints .......................................................... 4-18
4.5  Setting hardware breakpoints ............................................................. 4-24
4.6  Using complex breakpoints ................................................................. 4-30
4.7  Using the Break/Tracepoints pane ...................................................... 4-33
4.8  Disabling and clearing breakpoints ..................................................... 4-39
4.9  Saving breakpoints as favorites ......................................................... 4-41

Chapter 5  Memory Mapping
5.1  About memory mapping ................................................................. 5-2
5.2  Enabling and disabling memory mapping ........................................... 5-4
5.3  Setting up a memory map ................................................................. 5-5
5.4  Viewing the memory map ................................................................. 5-7
5.5  Editing map entries ......................................................................... 5-11
5.6  Setting top of memory and stack values .......................................... 5-12
5.7  Generating linker command files for non-ARM targets .................... 5-13

Chapter 6  Monitoring Execution
6.1  Working with registers .................................................................... 6-2
6.2  Working with memory .................................................................... 6-12
6.3  Working with the stack .................................................................... 6-23
6.4  Using the call stack ....................................................................... 6-28
6.5  Working with watches .................................................................... 6-32

Chapter 7  Reading and Writing Memory, Registers, and Flash
7.1  About interactive operations .............................................................. 7-2
7.2  Using the Memory/Register Operations menu .................................. 7-3
7.3  Accessing interactive operations in other ways ............................... 7-5
7.4  Working with Flash ....................................................................... 7-6
7.5  Examples of interactive operations .................................................. 7-12

Chapter 8  Working with Browsers
8.1  Using browsers ............................................................................. 8-2
8.2  Browsing modules and files ............................................................ 8-3
8.3  Browsing functions ....................................................................... 8-6
8.4  Browsing variables ...................................................................... 8-10
8.5  Specifying browser lists ................................................................. 8-13
8.6  Browsing C++ classes .................................................................. 8-15
8.7  Other routes to the browsers ......................................................... 8-17

Chapter 9  Working with Macros
9.1  About macros ............................................................................... 9-2
9.2  Using macros ............................................................................... 9-7
Chapter 10  Configuring Workspace Settings
10.1 Using workspaces ............................................................... 10-2
10.2 Viewing workspace settings ................................................ 10-8
10.3 Configuring workspace settings .......................................... 10-13

Chapter 11  Managing Projects
11.1 About managing projects .................................................... 11-2
11.2 Using the Project and Tools menus ...................................... 11-12
11.3 Managing your build tools .................................................. 11-18
11.4 Managing user-defined projects ......................................... 11-21
11.5 Creating a new user-defined project ................................. 11-29
11.6 Building your application ................................................... 11-41
11.7 Managing project properties ............................................. 11-46
11.8 Configuring project properties ........................................... 11-52
11.9 Managing build target configurations ............................... 11-72
11.10 Using the Project Control dialog box .............................. 11-81
11.11 Managing projects in the Process Control pane ............ 11-84
11.12 Project binding ............................................................... 11-90
11.13 Managing multiple projects ............................................ 11-102

Chapter 12  Editing Source Code
12.1 About the File Editor .......................................................... 12-2
12.2 Configuring the File Editor ............................................... 12-4
12.3 Using the Editor Buffer List ............................................... 12-6
12.4 Editing text ......................................................................... 12-9
12.5 Managing your editing session .......................................... 12-15
12.6 Using standalone editors ................................................... 12-19
12.7 Using templates ............................................................... 12-21
12.8 Working in vi mode ............................................................ 12-26

Chapter 13  Searching and Replacing Text
13.1 About finding and replacing text ....................................... 13-2
13.2 Searching and replacing text ............................................. 13-3
13.3 Searching multiple files ..................................................... 13-7
13.4 Working with functions ..................................................... 13-10
13.5 Pair matching ................................................................. 13-12
13.6 Configuring searches ......................................................... 13-15
13.7 Using regular expressions ................................................ 13-17

Chapter 14  Working with Version Control Systems
14.1 Defining the version control tool ....................................... 14-2
14.2 Using a version control tool .............................................. 14-3
Contents

14.3 Configuring a custom version control tool ................................................ 14-8

Appendix A Workspace Settings Reference
A.1 DEBUGGER ................................................................. A-2
A.2 CODE ................................................................. A-6
A.3 ALL ................................................................. A-9

Appendix B Project Properties Reference
B.1 PROJECT group ......................................................... B-3
B.2 PROJECT group for a Container project ............................................. B-8
B.3 SETTINGS group ......................................................... B-9
B.4 CONFIGURATION group .................................................. B-22
B.5 COMPILE group ......................................................... B-24
B.6 ASSEMBLE group ....................................................... B-46
B.7 CUSTOM group ......................................................... B-58
B.8 BUILD group .......................................................... B-60
B.9 BUILD_LIB group ........................................................ B-72
B.10 MAKEFILE group .................................................... B-80

Appendix C RealView Debugger on Solaris and Linux
C.1 About this Appendix ...................................................... C-2
C.2 Installing RealView Debugger ................................................ C-3
C.3 Getting more information ..................................................... C-9
C.4 Changes to target configuration details ........................................ C-10
C.5 Changes to GUI and general user information .............................. C-16
C.6 Connecting to remote hosts ................................................... C-24

Glossary
Preface

This preface introduces the RealView™ Debugger v1.6 User Guide that shows you how to use RealView Debugger to manage software projects and to debug your application programs. It contains the following sections:

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

RealView Debugger provides a powerful tool for debugging and managing software projects. This book contains:

- a detailed description of how to use RealView Debugger to debug applications programs, using a range of debug targets, including examples
- an explanation of the features of the RealView Debugger IDE so that you can manage your software projects and organize source files
- a description of the built-in features of RealView Debugger, such as workspaces and macros
- appendixes containing reference information for the software developer
- a glossary of terms for users new to RealView Debugger.

Intended audience

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

This book includes an appendix that contains information for developers using RealView Debugger on Solaris and Linux.

Before you start

It is recommended that you read RealView Debugger v1.6 Essentials Guide before starting to use this book. In particular, read the chapter describing the user desktop because this contains details about menus and GUI elements used in the rest of the documentation suite.

Examples

The examples given in this book have all been tested and shown to work as described. Your hardware and software might not be the same as that used for testing these examples, so it is possible that certain addresses or values might vary slightly from those shown, and some of the examples might not apply to you. In these cases you might have to modify the instructions to suit your own circumstances.

The examples in this book use the programs stored in the \Examples directory in your root installation, for example install_directory\RVD\Examples.
Preface

In general, examples use the ARMulator software simulator to simulate an ARM-based debug target. In some cases, examples are given for other debug target systems.

Using this book

This book is organized into the following chapters:

Chapter 1 Starting to use RealView Debugger
Read this chapter for details on how to start using RealView Debugger on your workstation.

Chapter 2 Working with Images
This chapter contains details on working with application programs in RealView Debugger, including how to load an image ready for debugging and how to view image details.

Chapter 3 Controlling Execution
Read this chapter for details of how to control program execution during your debugging sessions. It gives details on using the major control options and describes how to use files to keep a record of the debugging session.

Chapter 4 Working with Breakpoints
Read this chapter for details on using breakpoints to control execution of your application program. This chapter contains a full description of breakpoint options in RealView Debugger.

Chapter 5 Memory Mapping
This chapter gives details on managing memory for single processor operation during a debugging session. It describes the Process Control pane that contains a dynamic display of the current memory configuration.

Chapter 6 Monitoring Execution
Read this chapter for details of how to monitor execution of your application program by setting watches, reading registers and tracking changes to memory contents.
Chapter 7 Reading and Writing Memory, Registers, and Flash

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

Chapter 8 Working with Browsers

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

Chapter 9 Working with Macros

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

Chapter 10 Configuring Workspace Settings

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

Chapter 11 Managing Projects

RealView Debugger provides an integrated development environment for the organization and management of software projects that enable developers to share resources. These features are described in detail in this chapter.

Chapter 12 Editing Source Code

Read this chapter for a description of the file editor that is supplied as part of RealView Debugger.

Chapter 13 Searching and Replacing Text

Read this chapter for a description of using RealView Debugger to search and replace text in source files.

Chapter 14 Working with Version Control Systems

Read this chapter for a description of using the version control options in RealView Debugger.
Appendixes and Glossary

**Appendix A Workspace Settings Reference**
Read this appendix for details on setting options to configure your working environment using RealView Debugger workspaces. This appendix should be read in association with Chapter 10 Configuring Workspace Settings.

**Appendix B Project Properties Reference**
Read this appendix for details of how to configure your software projects. Read this appendix in association with Chapter 11 Managing Projects.

**Appendix C RealView Debugger on Solaris and Linux**
Read this appendix for details of how to use RealView Debugger on Solaris and Linux. This appendix contains corrections and additions to the documentation suite.

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

**Typographical conventions**
The following typographical conventions are used in this book:

*italic* Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

*bold* Highlights interface elements, such as menu names. Denotes ARM processor signal names. Also used for terms in descriptive lists, where appropriate.

*monospace* Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

*monospace* 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.
Further reading

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

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

ARM publications

This book is part of the RealView Debugger documentation suite. Other books in this suite include:
- *RealView Debugger v1.6 Essentials Guide* (ARM DUI 0181)
- *RealView Debugger v1.6 Command Line Reference Guide* (ARM DUI 0175)
- *RealView Debugger v1.6 Target Configuration Guide* (ARM DUI 0182)

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)

If you are using RealView Debugger on Solaris and Linux with RealView™ ARMulator® ISS v1.3, refer to the following documentation for more information:
- *RealView ARMulator ISS v1.3 User Guide* (ARM DUI 0207)
- *Addendum 01 RealView ARMulator ISS v1.3 Guide* (ARM DUI 0207).

The following documentation provides general information on the ARM architecture, processors, associated devices, and software interfaces:
Refer to the following documentation for information relating to the ARM debug interfaces suitable for use with RealView Debugger:

- *Multi-ICE® Version 2.2 User Guide* (ARM DUI 0048)

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


Refer to the following documentation for details on the FLEXlm license management system, supplied by GLOBEtrrotter 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:
Preface


For more information about the JTAG standard, see:


For more information about Oak and TeakLite processors from the DSP Group see:

Preface

Feedback

ARM Limited welcomes feedback on both RealView Debugger and its documentation.

Feedback on RealView Debugger

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

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

Feedback on this book

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

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

General suggestions for additions and improvements are welcome.
Chapter 1
Starting to use RealView Debugger

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

- Starting RealView Debugger on page 1-2
- Using RealView Connection Broker on page 1-6
- RealView Debugger directories on page 1-8.
1.1 Starting RealView Debugger

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

- Starting from Windows
- Starting from the command line
- Setting environment variables on page 1-4.

1.1.1 Starting from Windows

To start RealView Debugger:

1. Select Start → Programs → ARM RealView Debugger v1.6.1 from the Windows Start menu.

2. Select RealView Debugger from the menu.

1.1.2 Starting from the command line

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

```
rvdebug.exe [-bat|-cmd][-install=pathname][-user=name][-home=pathname]
[-aws=pathname][-aws=-][-exec image_pathname]
[-inc pathname][-jou pathname][-log pathname][-s pathnam][-nologo]
```

where:

- **-bat** Runs a RealView Debugger session in batch mode, that is without any user interaction.
  
  Use this with -inc to run a script file containing commands.
  
  Can be replaced with -b.

  ————  **Note**  ————
  
  Do not use -b without -inc. If you use only -inc, the script file is run with the GUI enabled.

- **-cmd** Runs the command-line debugger only to use CLI commands to carry out debugging tasks. This enables you to interact with the debugger without using the RealView Debugger GUI.

- **-install** Specifies the installation directory where this differs from the default installation. This is then used to define the location of the default RealView Debugger home directory when RealView Debugger runs for the first time.
  
  This must be used if the environment variable RVDEBUG_INSTALL is not set.
-user Specifies the user ID in the RealView Debugger home directory used for the debugging session. Where this is not specified, the default Windows login is used. See Defining the home directory on page 1-8 for details.

-home Specifies a RealView Debugger home directory used for the debugging session. If the specified directory does not exist, a new one is created. Where this is not specified, the default directory is used. See Defining the home directory on page 1-8 for details.

-aws Runs a RealView Debugger session with the specified workspace. This overrides any workspace specification that was stored when the previous session ended. Use -aws=- to start without a workspace.

-exec Specifies the image loaded when RealView Debugger runs. The image specification can also include target details and image arguments.

-inc Runs a RealView Debugger session with the specified include file. Use -inc:
   • in batch mode in association with the -bat setting, to execute the commands contained in the file and then exit the debugger
   • in command-line mode in association with the -cmd setting, to execute the commands contained in the file and then leave the debugger running ready to continue the debugging session
   • in GUI mode on its own, to execute the commands contained in the file during a debugging session.

-jou Runs a RealView Debugger session with the specified journal file open for writing. Can be replaced with -j.

-log Runs a RealView Debugger session with the specified log file open for writing. Can be replaced with -l.

-s Runs a RealView Debugger session with the specified STDIOlog file open for writing.

-no_logo Runs a RealView Debugger session without displaying a Windows splash screen.

Examples

To start RealView Debugger and specify an installation directory, where RVDEBUG_INSTALL is not set:
**Starting to use RealView Debugger**

```
install_directory\bin\rvdebug.exe -install="E:\Program Files\ARM\RealView Debugger"
```

If RVDEBUG_INSTALL is set then the -install overrides it.

To start RealView Debugger and specify a home directory, where RVDEBUG_HOME is not set:
```
install_directory\bin\rvdebug.exe -home="D:\Program Files\ARM\RealView Debugger\home\my_user_home"
```

To start RealView Debugger and specify a workspace:
```
install_directory\bin\rvdebug.exe" -aws="D:\Program Files\ARM\RealView Debugger\home\my_user_name\friday_test.aws"
```

To start RealView Debugger without loading a workspace:
```
install_directory\bin\rvdebug.exe -aws=
```

To start RealView Debugger with a log file open for writing:
```
install_directory\bin\rvdebug.exe -log "D:\Program Files\ARM\RealView Debugger\home\my_user_name\test_files\my_log.log"
```

To start RealView Debugger with a specified image loaded that takes two arguments:
```
install_directory\bin\rvdebug.exe -exec "C:\rvd\images\my_image.axf;;arg1 arg2"
```

In these examples, your install_directory might be a default location such as C:\Program Files\ARM\RVD\Core\1.6.1\81\win_32-pentium.

**Getting more information**

To find more information on operations available from the command line, see:
- Chapter 2 Working with Images for details on loading images.
- Chapter 3 Controlling Execution for details on using log and journal files.
- Chapter 10 Configuring Workspace Settings for details on workspaces.

### 1.1.3 Setting environment variables

User-defined environment variables can be set to configure RealView Debugger. Set RVDEBUG_INSTALL or RVDEBUG_HOME to override the default locations, for example to specify:
- an installation directory that differs from the default, set:
  ```
  RVDEBUG_INSTALL=D:\Program Files\ARM\RealView Debugger
  ```
Starting to use RealView Debugger

- a home directory that differs from the default, set:
  
  RVDEBUG_HOME=E:\Program Files\ARM\RealView Debugger\my_home

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

RVDEBUG_SHARE=H:\ournet\devel\rvd\shared
1.2 Using RealView Connection Broker

Execution vehicles can reside on the same workstation as RealView Debugger or any other workstation on your network. These services are handled by the RealView Connection Broker, rvbroker.exe.

RealView Connection Broker operates in two modes:

- **Local** Operating as RealView Connection Broker, this runs on your local workstation and enables you to access targets on the local workstation.

- **Remote** Operating as RealView Network Broker, this runs on a remote workstation and makes specified targets on that workstation available to other workstations connected to the same network.

Local host simulators are available immediately from the Connection Control window. If you expand the Simulator Broker entry, ready to connect to a simulator, RealView Debugger starts RealView Connection Broker in local mode to manage your connection.

1.2.1 Starting RealView Network Broker

Any remote workstation that is to give access to simulators or emulators must be running RealView Connection Broker in remote mode, that is RealView Network Broker. This can be started in two ways:

- if the remote workstation is running UNIX and the rsh command is available at the local workstation, the local workstation can start RealView Network Broker on the remote workstation
- if the remote workstation is running Windows, RealView Network Broker must be started explicitly on that workstation.

If you are using a remote Windows workstation to access simulators or emulators, start RealView Network Broker:

1. Log onto the remote workstation.
2. Select **Start** → **Programs** → **ARM RealView Debugger v1.6.1** from the Windows **Start** menu.
3. Select **RealView Network Broker**.

The syntax for the command-line method of starting RealView Connection Broker in remote mode is as follows:

```
rvbroker.exe -install=pathname -remote
```
where:

- `-install` Specifies the installation directory where this differs from the default installation.
  This must be used if the environment variable `RVDEBUG_INSTALL` is not set.

0          Specifies the TCP/IP port.

remote    Specifies that the TCP/IP port is used to make this workstation visible to other network users.

For example, to start RealView Network Broker on a Windows workstation and specify an installation directory, where `RVDEBUG_INSTALL` is not set:

```
rvbroker.exe -install="E:\ARM\RealView Debugger" 0 remote
```

Note

If you end a debugging session, and close down RealView Debugger, this does not terminate RealView Network Broker on the remote workstation. This must be shut down explicitly.

To access a remote host simulator or emulator using RealView Network Broker you must define the location of the remote workstation in your target configuration settings. The chapter describing configuring custom connections in RealView Debugger v1.6 Target Configuration Guide includes examples of how to set up your own connections.
1.3 RealView Debugger directories

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

- Defining the installation directory
- Defining the home directory
- Using the examples directories on page 1-9.

1.3.1 Defining the installation directory

RealView Debugger must be able to identify the installation directory so that it can locate user files and configuration files. It uses the following to define the installation directory (in order of priority):

1. The -install command line argument, where used.
2. The RVDEBUG_INSTALL environment variable, where set.
3. The default location as defined by the root installation.

1.3.2 Defining the home directory

RealView Debugger requires a home directory to store user-specific settings and configuration files. This is not the same as your Windows home directory. The location of this directory depends on the environment variables set, and the command line arguments used, when RealView Debugger starts. It uses the following tests to define the home directory:

1. The -home command line argument, if used.
2. The RVDEBUG_HOME environment variable, if set.
3. The -user command line argument, if used. This is then used to specify the user ID in the home directory, for example set USER=my_user_name to specify the home directory \home\my_user_name.
4. Your default Windows login, for example \home\WinLogID.

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

Because you can choose the home directory, the installation directory and your user name, the RealView Debugger home directory is defined in this book as being in a default location install_directory\home\user_name, where user_name is the Windows
login ID and install_directory is the default location for the base installation, such as C:\Program Files\ARM\RVD\Core\1.6.1\81\win_32-pentium. This means that your files might be stored in places other than those given in the examples.

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

1.3.3 Using the examples directories

Various demonstration projects are supplied as part of the RealView Debugger root installation. These contain programs in the form of ARM assembly language, C, or C++ source code files. These projects are stored in the \Examples directory in your root installation.

The root installation also includes demonstration projects, and associated files, for working with Flash. These are in \flash and \flash\examples.
Starting to use RealView Debugger
This chapter describes how to manage images during a debugging session. It contains the following sections:

- Loading images on page 2-2
- Managing images on page 2-8
- Working with symbols on page 2-16
- Working with multiple images on page 2-17
- Unloading and reloading images on page 2-19.
2.1 Loading images

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

RealView Debugger uses a board file to access information about the debugging environment and the debug targets available to you, for example how memory is mapped. See RealView Debugger v1.6 Target Configuration Guide for details of how to customize your targets.

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

Select File → Connection → Connect to Target... from the main menu to display the Connection Control window to make your first connection. For details on using this window, see the chapter describing getting started in RealView Debugger v1.6 Essentials Guide.

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

- Loading from a user-defined project
- Using the Load File to Target dialog box on page 2-3
- Loading from the Process Control pane on page 2-5
- Quick loading on page 2-5
- Loading from the command line on page 2-5
- Loading and runtime visualization on page 2-7.

The examples in this section assume that you are using a Typical installation and that the software has been installed in the default location. If you have changed these defaults, or set the environment variable RVDEBUG_INSTALL, your installation will differ from that described here.

2.1.1 Loading from a user-defined project

Where you have created a user-defined project, it is recommended that you open the project first to load and debug the associated image, or images. Opening the project enables you to access the project properties, save new settings, or make changes to the build model.
With a user-defined project open, for example \dhrystone\dhrystone.prj, from the \Examples directory in the root installation, click on the hyperlink in the File Editor pane to load the associated image.

Note

Loading an image built as part of a user-defined project without opening the project does not give you access to all the project properties because these are unknown to RealView Debugger. In this case, RealView Debugger creates an in-memory project, or uses the saved auto-project file (see Working with auto-projects on page 2-11 for details).

2.1.2 Using the Load File to Target dialog box

Select File → Load Image... from the Code window main menu to load an image to a processor for execution. This displays the Load File to Target dialog box shown in Figure 2-1.

![Figure 2-1 Load File to Target dialog box](image)

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

Symbols Only

By default, any object file loaded from this dialog box also loads the symbols. If you want to load only the symbols then select this check box, for example when you are working with ROM images.
If the program was initially compiled without a symbol table then you must recompile the program before loading only the symbols. See *Working with symbols* on page 2-16 for more details.

**Replace Existing File(s)**

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

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

**Target Name:**

Use this field to enter the target name, where supported.

A name entered here is then used as the argument to a `LOAD` command (see *Specifying the load instruction* on page 2-7).

**Arguments:**

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

Entries in this field create an arguments list used with the `LOAD` command (see *Specifying the load instruction* on page 2-7).

**PC**

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

**Auto-Set PC**

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

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

**Set PC to Entry point**

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

This is the default if you select both:

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

Unselect the **Set PC to Entry point** check box to prevent the load command setting the PC.
2.1.3 Loading from the Process Control pane

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

1. Select View → Pane Views → Process Control Pane from the default Code window main menu to display the Process Control pane. 
   Whilst there is no image loaded, the pane only shows details about the debug target processor and the current location of the PC.

2. Right-click on the top line, the Process entry, to display the Process context menu. 
   Whilst there is no image loaded, you can also display this menu from the Image entry.

3. Select Load Image... to display the Load File to Target dialog box.

4. Complete the entries in the dialog box, described in Using the Load File to Target dialog box on page 2-3, to load the required image.

2.1.4 Quick loading

You can load an image by dragging the appropriate executable file, with the .axf extension, and dropping it into the File Editor pane. If successful, this is the same as loading the image using the Load File to Target dialog box with the default settings (shown in Figure 2-1 on page 2-3), that is the load auto-sets the PC and overwrites any existing image on the debug target.

——— Note ————
Make sure that the current connection, as shown in the Code window title bar, matches the processor type of the image you are trying to load. If they do not match the load fails.

——— Note ————

2.1.5 Loading from the command line

You can start RealView Debugger from the command line and specify an image to load automatically. The syntax for loading an image this way is as follows:

rvdebug.exe -exec path

——— Note ————

Controls used here, for example setting the PC, take precedence over any load settings elsewhere.
where _pathname_ specifies the image loaded and can also include target details and image arguments (see *Specifying the load instruction* on page 2-7 for details).

If the _pathname_ includes spaces, it must be enclosed in quotes, for example:

```
"C:\Program Files\ARM\RVD\bin\rvdebug.exe" -install=C:\rvd
-exec "C:\rvd\my images\my_image.axf"
```

This starts RealView Debugger, specifies an installation directory, and issues a `load/pd/r` command to load the named image to your debug target. Any error messages appear in a dialog box, specified by `/pd`. This command replaces any image currently loaded on the chosen target, specified by `/r`.

--- Note ---

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

---

### Making a connection

If you are connected to your debug target, starting RealView Debugger in this way loads the specified image to the target and updates the Code window. This is the same as *Using the Load File to Target dialog box* on page 2-3.

If you are not connected to your debug target before starting RealView Debugger, loading an image from the command line starts the debugger and then displays a prompt box, shown in Figure 2-2, for you to complete the connection.

![Prompt](image)

**Figure 2-2 Connection prompt**

Click either:

- **Yes**
  
  Causes the debugger to wait until you connect to your debug target. The image is then loaded to the connected target.

- **No**
  
  Starts the debugger but cancels the image loading operation.
Specifying the load instruction

If you are loading an image from the command line, you can pass arguments to the image and specify the target name that is passed to the image loader. The syntax is as follows:

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

where:
- `image.axf` Specifies the image loaded.
- `target_name` Specifies the target name, where supported.
- `arg1 arg2 ...` Specifies an optional, space-separated, list of arguments to the image.

Specifying the target name depends on the underlying OS support and your debug target. For example, if you are using an RTOS image loader, then this target name is passed to the loader. In the example below, you are using the debugger built-in loader and so specifying target name has no effect and can be omitted:

```plaintext
C:\rvd\bin\rvdebug.exe -exec “C:\rvd\images\my_image.axf;;arg1 arg2 arg3”
```

--- Note ---

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

---

For details on debugging applications using an RTOS see the chapter describing RTOS support in *RealView Debugger v1.6 Extensions User Guide*.

2.1.6 Loading and runtime visualization

As you load an image to your debug target, the Code window Status line shows the progress of the load and gives an indication of the percentage complete.

The State group, on the Actions toolbar, shows the runtime state of the debug target. Where an image is loaded but not executing, this shows *Stopped*. A moving progress indicator signals an application is running.
2.2 Managing images

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

- Viewing image details in the Code window
- Viewing image details in the Process Control pane on page 2-9
- Working with auto-projects on page 2-11
- Working with user-defined projects on page 2-14.

The examples in this section assume that you are using a Typical installation and that the software has been installed in the default location. If you have changed these defaults, or set the environment variable RVDEBUG_INSTALL, your installation will differ from that described here.

2.2.1 Viewing image details in the Code window

If an image is successfully loaded to the target processor, the Code window is updated, shown in Figure 2-3.

![Code window with image loaded]

Figure 2-3 Code window with image loaded
When you load an image with symbols, as here, RealView Debugger searches for corresponding source files and displays these as tabs in the File Editor pane.

RealView Debugger updates the default views in the side pane and middle panes row and, where known, displays information about the new image. Because you have not started debugging, other panes are empty.

Click on the Src tab to display the source-level code view. The image was loaded with the Auto-set PC option set and so execution control is located at the default entry point. This is indicated by a box at line 78, colored red by default.

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

### 2.2.2 Viewing image details in the Process Control pane

Select View → Pane Views → Process Control Pane to display the Process Control pane, shown in Figure 2-4.

![Figure 2-4 Image details in the Process Control pane](image)

The Process Control pane contains tabs:

- **Process**: Displays details of the target processor or, in multiprocessor debugging mode, the current process.  
  See Working with processes on page 2-10 for details.

- **Map**: Displays the memory mapping for the target processor, or the current process, to enable you to change the map settings.  
  See Chapter 5 Memory Mapping for details on using this tab.
The tabs displayed in the Process Control pane depend on the debugging mode that you are licensed to use and your current debugging environment. For example, when debugging multithreaded applications, a **Thread** tab is displayed. See the chapter describing multiprocessing in *RealView Debugger v1.6 Extensions User Guide* for more details.

**Working with processes**

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

- image name
- image resources, including DLLs
- how the image was generated
- load parameters
- associated files
- execution state.

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

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

In the example in Figure 2-4 on page 2-9, you can see the entries:

**Current process**

Shows the target processor and the current state of any running process.

Where the process is stopped, as here, this shows the location of the PC. Where the process is executing, this changes to **run**.

**Image**

Details the loaded images:

**Load**

For each image, a check box indicates the load state and what has been loaded, that is image, symbols, or both.
Project  Shows that the project associated with the connection is either a real, user-defined project file (shown by the project name) or an auto-project (shown by <Auto>).

Settings  Shows where project settings are stored. These might be from a disk file (shown by <Saved>) or from an in-memory auto-project (shown by <Not Saved>).

Sources  These are either the sources making up the project, sources extracted from the makefile used in the build, or sources from the loaded image. Depending on the type of project, right-click on this entry to display a context menu to specify how sources are collected.

Getting more information about an entry in the Process tab

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

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

2.2.3 Working with auto-projects

An auto-project is a custom, image control, project that holds project settings where the build model is unknown.

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

If, for example, you load the image C:\demo_ARM\dhrystone\Debug\dhrystone.axf, RealView Debugger looks for the corresponding auto-project file C:\demo_ARM\dhrystone\Debug\dhrystone.axf.apr. Where no auto-project exists, RealView Debugger creates an in-memory auto-project, named dhrystone. The Process tab is then updated with the project details, shown in Figure 2-5 on page 2-12.
RealView Debugger gives you the option to save the in-memory settings to a file to use next time the image is loaded or as the basis of a new user-defined project.

**Viewing project settings**

You can view settings for the in-memory auto-project just like a user-defined project:

1. Right-click on the Project entry, to display the Project context menu.
   You can also display this menu from the Settings entry.

2. Select **Project Properties**... to display the Project Properties window where you can view the project settings. These are derived from the image details or created using defaults by RealView Debugger.

3. Select **File → Close Window** to close the Project Properties window without making any changes.

**Changing project settings**

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

**Note**

Changing auto-project settings might not take effect until the next time the image is loaded and executed. Reload an image to implement any new settings.
You can change settings for the in-memory auto-project just like a user-defined project:

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

![Figure 2-6 Changing auto-project settings](image)

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

4. Expand the **SETTINGS** group to see the image settings.
   Figure 2-6 shows the **Image_load** group and other image settings, such as breakpoints and runtime controls.
5. Right-click on the **Image_load** group and select **Explore** to see the group contents in the right pane.
6. Right-click on the **Set_pc** entry and select **never** from the options.
7. Select **File → Save and Close** to save your changes and close the Project Properties window.

To return the setting to the default:

1. Display the Project Properties window.
2. Right-click on the entry to display the context menu.
3. Select **Reset to Default** to restore the setting.
4. Select **File → Save and Close** to save your changes and close the Project Properties window.
See Appendix B *Project Properties Reference* for details on the settings shown here.

### Saving project settings

Save the auto-project so that the new settings are used when you next load the image. There are two ways to save an auto-project:

- In the Project Properties window, select **File → Save Changes** to close the window and save any changes to the file `dhrystone.axf.apr`.

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

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

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

### Closing auto-projects

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

**Note**

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

See Chapter 11 *Managing Projects* for more details on working with auto-projects.

### 2.2.4 Working with user-defined projects

With a user-defined project open, right-click on the *Project* entry to display the **Project** context menu, shown in Figure 2-7 on page 2-15.
Figure 2-7 Project menu

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

See Chapter 11 *Managing Projects* for full details on these options.

**Closing user-defined projects**

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

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

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

If you do not unload the image, the debugger:

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

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

---

**Note**

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

---

See Chapter 11 *Managing Projects* for more details on working with user-defined projects.
2.3 Working with symbols

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

If you select the Symbols Only check box, on the Load File to Target dialog box (see Figure 2-1 on page 2-3), the symbolic references are loaded into the debugger without loading any code or data to the target. You might want to do this if the code and data are already present on the debug target, for example in a ROM device.

You can choose to refresh the symbol data for a loaded image during your debugging session. There are two ways to do this for the current process, depending on the number of images loaded:

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

Note

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

RealView Debugger provides the option to load multiple images to the same debug target, that is where there is only a single connection. This enables you to load, for example, both an executable image and an RTOS at the same time.

To load two images to the same debug target:
1. Load the first image, for example hello.axf in the usual way.
2. Load a second image to the same target, for example demo.axf. The two images must not overlap in memory.
   
   ______ Note ______
   Remember to unselect the Replace Existing File(s) check box.

   ______

3. Select View → Pane Views → Process Control Pane from the default Code window main menu to display the Process Control pane.
4. Expand the display to see the process details, shown in Figure 2-8.

![Figure 2-8 Multiple images in the Process Control pane](image)

In this example, RealView Debugger:
- creates an in-memory auto-project for hello.axf
- binds hello.axf.apr to the current connection (using default binding)
- creates an in-memory auto-project for demo.axf
- does not bind demo.axf.apr, because there is no connection available
- updates the Code window title bar to show the active project (hello)
- updates the title bar of the floating Process Control pane to match.

For information on projects and project binding see Chapter 11 Managing Projects.
Because neither image is currently executing, the Process entry shows the current location of the PC, auto-set when the first image was loaded. You can move the PC manually to start debugging or reload the image that you want to test. For information on changing the PC see:

- Chapter 3 *Controlling Execution* for details on setting scope.
- Chapter 6 *Monitoring Execution* for details on changing register entries.

**Note**

If you are working with multiple images, you must set a manual breakpoint at the entry point for any images loaded above 0x8000. This ensures that RealView Debugger is able to debug these images in the usual way.

See the chapter describing multiprocessing in *RealView Debugger v1.6 Extensions User Guide* for more details on working with multiprocessing applications in the Process Control pane.
2.5 Unloading and reloading images

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

- Resetting your target processor
- Unloading an image
- Reloading an image on page 2-20.

2.5.1 Resetting your target processor

Where supported by your debug target, you might want to reset your target processor during a debugging session. The reset might be hard or soft depending on the processor type. See your processor hardware documentation for details.

If the processor chosen for reset has an image loaded then this might be unloaded on reset. The image can be reloaded as described in Reloading an image on page 2-20.

To reset a processor:

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

2.5.2 Unloading an image

You do not have to unload an image from a debug target before loading a new image for execution. To load over an existing image, ensure that the Replace Existing File(s) check box is checked on the Load File to Target dialog box, shown in Figure 2-1 on page 2-3. This automatically removes all details about the first image from RealView Debugger.

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

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

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

If there are any source file tabs currently displayed in the File Editor pane, the Src tab is brought to the top automatically when you unload because there is no known context. The open files remain available for further editing.
Unloading an image does not affect target memory. It unloads the symbol table and removes debug information from RealView Debugger. However, the image name is retained for reloading and the associated auto-project, or user-defined project used to load the image, is maintained.

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

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

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

However, if you close the project explicitly, you will have to load the image again after you reconnect because all image details have been removed.

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

- load the previously unloaded image
- reload the image.

Reloading refreshes any window displays and updates debugger resources.

To reload an image:

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

An image that has been unloaded cannot be reloaded using this option. Instead you must load the image again using File → Load Image....
• Click the **Reload Image** button from the Actions toolbar.

• Select the **Load** check box in the Process Control pane. This is the same as double-clicking on the Load entry.

RealView Debugger uses auto-project settings to load an image and these are automatically used when you reload the image, or when you select it from the Recent Images list.
Working with Images
Chapter 3
Controlling Execution

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

- Submitting commands on page 3-2
- Defining execution context on page 3-3
- Using Execution controls on page 3-6
- Working with the Debug menu on page 3-9
- Controlling debugging on page 3-13.
3.1 Submitting commands

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

If an application is currently executing, RealView Debugger uses a command queue to handle those commands that cannot be executed immediately. Through this mechanism, commands build up on the queue and are then executed when resources become available. Commands are never executed out of order.

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

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

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

The following notes apply to the command queue:

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

Click the Cancel button on the Actions toolbar to clear, or purge, the most recent pending command.
3.2 Defining execution context

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

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

3.2.1 Code views

Use the File Editor pane to view source code during your debugging session. In the example shown in Figure 2-3 on page 2-8, the File Editor pane contains three tabs:

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

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

If you click on a tab, for example the **Src** tab, and the statement where the PC is located is in view, a red box is drawn around that statement to highlight it in the chosen code view.

3.2.2 Defining scope and context

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

When your program is executing, the PC stores the address of the current execution point. By default, the scope is set when the PC changes. Loading an image sets the PC at the entry point using *autoscope*, that is the PC defines the scope. Autoscope is also used in an assembly language routine when you step into code that has no source information. In this case, RealView Debugger shows the last calling function that had valid source in the **Src** tab.

RealView Debugger uses a red box to highlight the location of the PC where this is visible in the selected code view. The PC is only visible in an execution tab. However, if you force scope to a different location then the red box highlights the current context. This might not be the true location of the PC.
When RealView Debugger first loads an image, and assuming that you do not force scope, the File Editor pane contains tabs showing program execution:

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

**Locating the Program Counter**

You can locate the PC using the **Debug** menu:

- select **Debug** → **Execution Control** → **Show Line at PC** to display the current location of the PC in the Output pane
- select **Debug** → **Execution Control** → **Show Context of PC** to display the current context of the PC in the Output pane.

**Forcing scope**

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

1. Connect to your target and load an image, for example dhrystone.axf.
2. Click on the **Src** tab to view the source file dhry_1.c.
   The PC is at the entry point at line 78, marked with a red box.
3. Right-click at a location (line or address) in your execution view, for example line 149 in the source file dhry_1.c.
4. Select **Scope To Here** from the context menu.
   The forced scope is identified by a filled blue pointer at the chosen location. This moves the red box to highlight the current context.
5. Select **Debug** → **Execution Control** → **Show Context of PC** to see the location of the PC in the Output pane.
6. Click **High-level Step Into** to step into the program.
7. Select **Debug** → **Execution Control** → **Show Context of PC** to see the location of the PC.

To reset the PC to the entry point, either:

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

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

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

- **Using the Execution group**
- **Using shortcuts** on page 3-8.

**Note**

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

### 3.3.1 Using the Execution group

The **Execution** group contains:

**Go**

Click this button to start from the current location of the PC and run until the program:
- ends
- encounters an error condition
- reaches a breakpoint
- reaches a halt condition.

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

**High-level Step Into**

Click this button to step by lines of source code until the PC is located in another function or context. Normally, click this button to step up by one call level, but, depending on the function stepped into, it might cause the program to execute until it reaches source code.

If the source line makes a function call, RealView Debugger steps into the function, unless there is no source code available for this function.

To see an example using this option:

1. Connect to your target and load an image, for example `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Set a simple breakpoint by double-clicking on line 78.
4. View the Output pane message:
5. Click **Go** and the program begins execution and runs up to the breakpoint.
   The Output pane shows where execution stops.

6. Click **High-level Step Into** once. A red box shows the location of the PC at line 91.

7. Click **High-level Step Into** several times to move through the lines of source code.

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

   **Note**
   You can get to `main()` using **Go** (as above) or a single step. The single step executes (from the high-level code) up to `main()`.

---

**High-level Step Over**

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

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

**Low-level Step Into**

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

**Low-level Step Over**

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

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

**Go until Return**

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

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

**Note**

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

### 3.3.2 Using shortcuts

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

Keyboard shortcuts are available, shown in the **Debug** menu, depending on the current code view. These are summarized in Table 3-1.

**Table 3-1 Keyboard shortcuts**

<table>
<thead>
<tr>
<th>Code view</th>
<th>Function key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>F10</td>
<td>Step by line over functions</td>
</tr>
<tr>
<td>Source</td>
<td>F11</td>
<td>Step by line into functions</td>
</tr>
<tr>
<td>Disassembly</td>
<td>F10</td>
<td>Step by instruction over functions</td>
</tr>
<tr>
<td>Disassembly</td>
<td>F11</td>
<td>Step by instruction into functions</td>
</tr>
</tbody>
</table>
3.4 Working with the Debug menu

Debugging your application programs relies on being able to control the execution of your code on the debug target. You must then be able to examine the contents of memory, registers, or variables, possibly continue execution one instruction at a time, or specify other actions to examine in detail the execution history. The Code window main menu includes the Debug menu, shown in Figure 3-1.

This section describes the Debug menu providing a starting point for many debugging operations:

- Using the Debug menu
- Using the Execution Control menu on page 3-10.

3.4.1 Using the Debug menu

The Debug menu offers the options:

**Execution Control**

This enables you to control how your program executes. You can run a series of instructions, step through instructions one at a time, or specify conditions that must be met to stop execution.

For details on using this option in your debugging session see Using the Execution Control menu on page 3-10.

**Simple Breakpoints**

This provides access to the Simple Breakpoints menu to use breakpoints in your code.

**Complex Breakpoints**

This provides access to the Complex Breakpoints menu to use breakpoints in your code. You must use a debug target that supports these breakpoints.
Tracepoints

This enables you to set tracepoints so that trace data can be captured. This is enabled if you have the appropriate Trace license. See the chapter describing tracing in *RealView Debugger v1.6 Extensions User Guide* for full details.

Memory/Register Operations

This enables you to examine memory and register contents during execution and to edit these interactively. In this way you have complete control over the way your application program runs.

Include Commands from File...

This enables you to store debugger commands in the form of a script file and then run the file to automate the debugging process.

Set Source Search Path...

This enables RealView Debugger to find sources automatically. This is only necessary if the compiler or assembler does not pass source paths.

Add/Edit Debugger Macros...

This enables you to use the RealView Debugger macro facility to create macros containing complex procedures. You can attach a macro to a breakpoint or run it as a command.

### 3.4.2 Using the Execution Control menu

The **Debug** menu enables you to define the execution path using the **Execution Control** menu, shown in Figure 3-2. In most cases, you use the menu options in combination with other debugging tools such as breakpoints.

<table>
<thead>
<tr>
<th>Function</th>
<th>Key Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset Target Processor</td>
<td>F5</td>
</tr>
<tr>
<td>Go to Start Execution</td>
<td>Del+F10</td>
</tr>
<tr>
<td>Go to Quiescence</td>
<td>Shift+F11</td>
</tr>
<tr>
<td>Step Into</td>
<td>F11</td>
</tr>
<tr>
<td>Step Over (load)</td>
<td>F10</td>
</tr>
<tr>
<td>Step Until Condition...</td>
<td>Del+Shift+F11</td>
</tr>
<tr>
<td>Step Execution</td>
<td>Del+Shift+Pause</td>
</tr>
<tr>
<td>Show Line at PC</td>
<td>Alt+F10</td>
</tr>
<tr>
<td>Show Context at PC</td>
<td>Alt+Shift+F10</td>
</tr>
<tr>
<td>Toggle Source/Disassembly</td>
<td>Del+F11</td>
</tr>
</tbody>
</table>

*Figure 3-2 Execution Control menu*
The **Execution Control** menu options are grouped according to their impact on the execution path:

**Reset Target Processor**
- Performs a processor reset operation on the current connection.
- The reset might be hard or soft depending on the processor type, see your processor hardware documentation for details.
- Simulating a processor reset does not reset variables to their defaults because memory is not re-initialized. The PC is reset.

**Go (Start Execution)**
- With an image loaded to the target processor, select this option to run the program, starting from the current location of the PC.

**Go to Cursor**
- With an image loaded, you can scroll down the listing and position the cursor on a specific line. Select **Go to Cursor** to execute the program up to the temporary breakpoint at the cursor, assuming that no halting condition occurs first.
- If you select the **Src** tab, the line marked by the cursor is enclosed in a red box indicating the position of the PC.
- Click **Go**, or use a **Step** control, to resume execution.

**Go until Return**
- Select this option to run from the current PC until control returns from the current function.
- Selecting this option stops execution at the assembler instruction immediately after the return, and not the next statement. If you select the **Src** tab, this has the effect that the red box indicating the position of the PC might be still located at the function call.

**Step Into**
- Steps execution, by lines of source code or assembler instructions, into functions. This behavior depends on the current code view, that is whether you have selected the **Src** tab or the **Dsm** tab.

**Step Over (next)**
- Steps execution, by lines of source code or assembler instructions, over functions. This behavior depends on the current code view, that is whether you have selected the **Src** tab or the **Dsm** tab.
Controlling Execution

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

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

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

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

**Show Line at PC**
Select this option to report the current module and procedure scope.

**Show Context of PC**
Select this option to report the current context showing the current root, module, procedure, and line.

**Toggle Source/Disassembly**
Select this option to toggle between the source-level view and the disassembly-level view in the File Editor pane.
3.5 Controlling debugging

The **Debug** menu includes options that give you control over debugger operations:

- **Using include files**
- **Searching for source files** on page 3-16.

3.5.1 Using include files

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

During your debugging session, you can create a log file of all the commands you enter. This file can then be used as the basis of a command file or a macro. By default, log files have the extension `.log` or `.inc`, but you can use any extension for writing.

Another log file, called the `STDIOlog` log file, enables you to keep a record of debuggee output only, that is messages from the target. This might be useful for controlling debugging by running scripts without using the RealView Debugger user interface. By default, these files have the extension `.log`, but you can use any extension for writing.

In addition to the log files, you can also create a journal of a debugging session. The session journal file you create contains all information including your commands, debugger output and any messages displayed in the Output pane. By default, journal files have the extension `.jou`, but you can use any extension for writing.

In summary:

- log files record commands you enter and messages from the debugger
- STDIO log files record messages from the debuggee only
- journal files record commands you enter and messages from the debuggee.

The following sections describe how to manage your debugging session using include files such as log or journal files:

- **Output buffering**
- **Creating log and journal files** on page 3-14
- **Closing log and journal files** on page 3-15
- **Including files** on page 3-15.

Output buffering

In the current release of RealView Debugger, output to the log files and journal files is unbuffered. This means that all lines are immediately flushed to the specified file. To change this, so that output to a file is buffered, set the `JOULOG_UNBUF` environment variable.
Creating log and journal files

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

To create an output file, or to open an existing file for writing:

1. Select **File → New → Log File...** to display the Select File to Log to dialog box where the file can be located.

2. Specify the pathname of the new log file, or locate a file created previously, for example `\home\my_user_name\my_log.log`.

3. Click **Save** to confirm the settings and close the dialog box.

4. If the specified log file already exists, RealView Debugger displays the File Exists prompt.

   This gives you the options:
   - **Yes** appends new commands to those already saved in the file
   - **No** replaces, or overwrites, any commands already saved in the file
   - **Cancel** closes the prompt and discontinues the log file access.

   Click **Yes** or **No** as required.

Output is now recorded automatically in the specified file.

RealView Debugger shows that it is recording using the status display area at the bottom of the Code window, shown in Figure 3-3.

![Figure 3-3 Output files recording](image)

Using log and journal files at start-up

You can start RealView Debugger and open a log or journal file for writing. Do this from MS-DOS, or from a Command Prompt window, or create a desktop shortcut, for example:

`rvdebug.exe -s "C:\RealView Debugger\test_files\my_image_file.log"`

If the file does not exist, RealView Debugger creates it. Where the file exists, RealView Debugger overwrites the current contents, without displaying a warning message.
When RealView Debugger starts to write to the log file, it records the filename as the first entry, for example:

;;;LOG FILE: C:\RealView Debugger\test_files\my_log.log

### Closing log and journal files

If you are recording a log, or journal file and you try to start a new recording, RealView Debugger gives you the option to close the current file so that a new file can be used.

To close a log or journal file, select **File → Close Logs/Journals...**. This displays a list selection box, shown in Figure 3-4, where you can specify which file, or files, to close.

![Figure 3-4 Close log and journal files selection box](image)

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

- **OK**  
  Click this button to close selected files and then close the selection box.

- **Cancel**  
  Click this button to leave all files open and then close the selection box. Using **Cancel** ignores the status of any check boxes in the list.

- **Help**  
  Click this button to display the online help for this selection box.

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

### Including files

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

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

3. Click **Open** to load the file and execute the commands stored there.

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

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

### 3.5.2 Searching for source files

By default RealView Debugger searches for application source file paths according to information contained in the image. If no paths are provided in the program image file, RealView Debugger looks in the current working directory for the source file or files.

If the default search pattern fails, RealView Debugger uses the stored search paths.

From the Code window, select **Debug → Set Source Search Path...** to display the Source Search Paths dialog box, shown in Figure 3-5, where you can specify the search paths.

![Figure 3-5 Source Search Paths dialog box](image)

On first opening, this dialog shows any paths already set by the RVDEBUG environment variable. Use the directory button to locate directories where source files are stored and add their pathnames to the list.

Where a path is checked, it is included in any search. If you use the directory button to add a pathname to the list, it is automatically checked for inclusion. To exclude a pathname from the search, click on it so it is unselected.
Note

If you have used your project to store search paths, these are displayed in the Source Search Paths dialog box, and override any paths defined by the RVDEVUG environment variable.

Use the controls to manage pathnames:

**Add** Adds the pathname specified in the data entry field, at the top of the dialog box, to the list. A pathname added to the list in this way is automatically checked, that is it is included in the search.

Any new files that you add to the list in the dialog box are not saved when the current session ends.

**Del** Deletes checked entries from the displayed list.

**Rep** With a pathname entered into the data entry field and a single pathname checked, use this button to replace the checked item with the chosen pathname. A pathname replaced in this way is not automatically checked. Select the entry so that it is included in the search. You can only replace one item at a time in the list.

**AllOn** Enables all the items in the pathname list so that every item is checked.

**AllOff** Disables all items in the pathname list so that every item is unselected.

With the required search paths set up, click OK to confirm your choice and close the dialog box. Pathnames that are checked reappear when the dialog box is next displayed in this session.

Click Cancel to abort any changes to the specified paths and close the dialog box.
Chapter 4
Working with Breakpoints

This chapter explains the different types of breakpoints supported by RealView Debugger, describes the options for setting breakpoints, and explains how to manage breakpoints during your debugging session. It includes the following sections:

- Breakpoints in RealView Debugger on page 4-2
- Setting breakpoints quickly on page 4-5
- Using simple breakpoints on page 4-10
- Setting conditional breakpoints on page 4-18
- Setting hardware breakpoints on page 4-24
- Using complex breakpoints on page 4-30
- Using the Break/Tracepoints pane on page 4-33
- Disabling and clearing breakpoints on page 4-39
- Saving breakpoints as favorites on page 4-41.
4.1 Breakpoints in RealView Debugger

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

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

When a breakpoint triggers, RealView Debugger can carry out higher level requests. For example you can:
- attach macros to breakpoints
- update windows or files
- change the behavior of your application program.

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

This section describes:
- Breakpoint types
- Using hardware breakpoints on page 4-3.

4.1.1 Breakpoint types

RealView Debugger enables you to use different types of breakpoint. What is available depends on the:
- hardware support provided by your target processor
- connection vehicle used to maintain the target connection.

In RealView Debugger there are two types of breakpoint:

**Simple**

Simple breakpoints enable you to test address-specific data values. These breakpoints can be either hardware or software breakpoints.

When debugging code in ROM, RealView Debugger implements a simple hardware breakpoint by default. However, this depends on the hardware characteristics of your target processor.

Simple breakpoints are supported by all ARM processors.
Complex breakpoints use advanced hardware support on your target processor, or as implemented by your simulator software. Where supported by your debug target, complex breakpoints might be data-dependent or take advantage of range functionality, for example two breakpoints can be coupled together.

Check your hardware characteristics, and your vendor-supplied documentation, to determine the level of support for complex breakpoints (see Viewing your hardware characteristics for details).

### 4.1.2 Using hardware breakpoints

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

RealView Debugger sets a hardware breakpoint where possible if you request an instruction break in ROM. However, to use hardware breakpoints, your debug target must include support for such breakpoints. Even where this support is available, your target might be limited in the number it can support at one time. RealView Debugger menu options related to hardware breakpoints are grayed out if your target cannot support them or if no more are available.

**Note**

If you are using an ARMulator to simulate a target processor then no hardware support is available.

**Viewing your hardware characteristics**

To see your hardware support for breakpoints select **Debug → Complex Breakpoints → Show Break Capabilities of HW...** from the default Code window main menu. This displays an information box describing the support available for your target processor, shown in Figure 4-1 on page 4-4.
Figure 4-1 shows the results for a core that contains hardware extensions for advanced debugging operations. In this example, the target processor is able to support a large number of hardware breakpoints.
4.2 Setting breakpoints quickly

You can set a breakpoint directly from the default Code window, depending on your current code view. This is useful to set a quick test point during a debugging session. This section describes:

- Quick breakpoints in source-level view
- Quick breakpoints in disassembly-level view on page 4-6
- Viewing breakpoints in your code view on page 4-7
- Clearing breakpoints quickly on page 4-8
- Setting breakpoints in other ways on page 4-9.

4.2.1 Quick breakpoints in source-level view

Double-click in the gray area to the left of a line to set a breakpoint quickly, shown in Figure 4-2. You can also double-click on the line number if this is visible.

![Figure 4-2 Setting a breakpoint quickly on a line](image)

This sets a simple breakpoint marked by a red disc in the margin.

If you want to set a breakpoint on a symbol, right-click on the symbol to display the context menu, shown in Figure 4-3 on page 4-6, and select **Set Break On**.

If you want to set a breakpoint on a multi-statement line, for example on the `for`...`loop`, right-click on the statement to display the context menu shown in Figure 4-3 on page 4-6, and select **Set Break on Statement**.
Working with Breakpoints

Figure 4-3 Setting a breakpoint quickly on a statement

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

```
b1 \DHRY_1\ #146:3
```

Setting breakpoints in source-level view inserts a software instruction breakpoint by default. This is set using a `BREAKINSTRUCTION` command. RealView Debugger attempts to set a software breakpoint where the code is not in ROM.

If your code is in ROM, RealView Debugger sets a hardware breakpoint where one is available. An error message is displayed if no such breakpoint is available.

See Using the Break/Tracepoints pane on page 4-33 for details on viewing breakpoints in the Break/Tracepoints pane.

4.2.2 Quick breakpoints in disassembly-level view

Double-click on the required instruction, on the Dsm tab, to set a breakpoint quickly, shown in Figure 4-4.

Figure 4-4 Setting a breakpoint quickly on an instruction
To set a breakpoint on the destination of a branch instruction:

1. Locate the required instruction in the Dsm tab.
2. Right-click to display the context menu, shown in Figure 4-5.
3. Select Set Instr Break.

![Figure 4-5 Setting a breakpoint quickly on a branch instruction](image)

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

   \[ \text{bi} \ 0x8064 \]

As with the source-level view, RealView Debugger sets a software or hardware breakpoint depending on where your program is stored and what breakpoints are available.

See Using the Break/Tracepoints pane on page 4-33 for details on viewing breakpoints in the Break/Tracepoints pane.

### 4.2.3 Viewing breakpoints in your code view

Breakpoints are marked in the source-level and disassembly-level view at the left side of the window using color-coded discs:

- **Red**: This symbol shows the position of an enabled breakpoint. A second breakpoint cannot be set at the same location as an existing breakpoint.
Working with Breakpoints

**Yellow** If you set a conditional breakpoint, that is one that stops execution when certain conditions are met, the marker is a disc filled with yellow.

**White** Where a breakpoint has been set previously and then disabled, it is marked by a white disc. If the breakpoint is re-enabled the disc changes color as appropriate for the type of breakpoint.

If you have set multiple breakpoint units, for example on a source line containing multiple statements or on inlined functions, then disabling the first breakpoint unit changes the marker disc, shown in Figure 4-6. If you disable the second breakpoint unit on the line, the marker remains.

![Figure 4-6 Working with breakpoint units](image)

See *Using the Break/Tracepoints pane* on page 4-33 for details on viewing breakpoints in the Break/Tracepoints pane.

If you try to set a breakpoint on a non-executable line, RealView Debugger looks for the first executable line immediately following and places the breakpoint there. If the lines preceding the breakpointed instruction are comments, declarations, or other non-executable code, they are marked with black, downward pointing arrows. Lines marked in this way are regarded as part of the breakpoint. You cannot place two unconditional breakpoints on the same line, or on lines marked by the downward pointing arrows.

If you have set tracepoints, these are shown as green right-facing arrows in the current code view. Similarly, thread-specific breakpoints are shown as green stop signs. See *RealView Debugger v1.6 Extensions User Guide* for details of using these features.

### 4.2.4 Clearing breakpoints quickly

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

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

4.2.5 Setting breakpoints in other ways

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

- Use drag-and-drop to create a simple breakpoint in the Break/Tracepoints pane based on a program item, for example highlight a source code function in the File Editor pane and then drag it (using your mouse) and drop it into the Break/Tracepoints pane (see Using the Break/Tracepoints pane on page 4-33 for details).

- Set a breakpoint on a memory location in the Memory pane. The type of breakpoint offered depends on the type of memory at the chosen location, for example RAM, ROM, or Flash.

- Set a hardware breakpoint at an address of a symbol using the Stack pane. However, this is not recommended if execution runs past the end of a function return call because as soon as you exit the function the stack value is no longer meaningful.

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

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

- Set a conditional breakpoint on a value shown in the Watch pane.
4.3 Using simple breakpoints

The Debug menu provides a range of breakpoint options to use during your debugging session. Select Debug → Simple Breakpoints from the Code window main menu to display the Simple Breakpoints menu, shown in Figure 4-7.

This menu offers different breakpoint operations:

- Using the Set Address/Data Break/Tracepoint dialog box
- Setting simple conditional breakpoints on page 4-13
- Breakpoint operations on page 4-16
- Setting breakpoints from saved lists on page 4-17.

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

Select Debug → Simple Breakpoints → Address/Data... from the Code window main menu to display the Set Address/Data Break/Tracepoint dialog box, shown in Figure 4-8 on page 4-11.
The Set Address/Data Break/Tracepoint dialog box provides comprehensive facilities to enable you to specify new breakpoints in full. You can also use it to edit existing breakpoints.

The main interface components of the Set Address/Data Break/Tracepoint dialog box are:

**Location**
Specifies the memory location where the new breakpoint is set. Click the drop-down arrow to the right of this field to choose from a list browser, or select from your personal favorites list, or select from a list of previously-used expressions. The options shown here depend on your debug target and connection.

Where supported by your target hardware, use the options from the right-arrow menu to qualify the location (see *Using ranges and masks* on page 4-27 for details).

This field is enabled if you select a suitable Breakpoint Type and your current target supports the chosen type.

**Value Match**
Enter the data value that triggers the breakpoint. Click the drop-down arrow to the right of this field to choose from a list browser, or select from your personal favorites list, or select from a list of previously-used expressions.
If you use this with data breakpoints, this compares the data value that is read or written.

Where supported by your target hardware, use the options from the right-arrow menu to qualify the value match (see Using ranges and masks on page 4-27 for details).

This field is enabled if you select a suitable Breakpoint Type and your current target supports the chosen type.

**Break/Tracepoint Type**

Enables you to select the type of breakpoint to set. On first opening the dialog box, the list shows only the breakpoint types that are supported by your debug target.

If you choose a breakpoint type, this changes the contents of the groups in this dialog box.

**HW Support**

This area is populated if you select a suitable Breakpoint Type and your current target supports the chosen type.

Where your debug target supports breakpoint tests in hardware, they can be managed and edited using this group. If enabled, the display lists currently available tests.

See Setting hardware breakpoints on page 4-24 for details on using these controls.

**Qualifiers**

When setting a conditional breakpoint, you specify the condition that must be satisfied to trigger the breakpoint. Qualifiers are the tests that RealView Debugger carries out to trigger the breakpoint. Click **New** to display the New Qualifiers menu, shown in Figure 4-9, where you can define the test criteria.

![Figure 4-9 Breakpoint New Qualifiers menu](image)

**Figure 4-9 Breakpoint New Qualifiers menu**

See Setting conditional breakpoints on page 4-18 for details on using these controls.

**Actions**

When a conditional breakpoint triggers the usual action is to stop execution but you can specify one or more debugger actions that must be performed when execution stops. In addition, RealView Debugger can...
carry out the specified action and then execution can continue. This is useful when debugging complex applications without direct user intervention.

Click **New** to display the **New Actions** menu, shown in Figure 4-10.

**Figure 4-10 Breakpoint New Actions menu**

See *Setting conditional breakpoints* on page 4-18 for details on using these controls.

**OK**
Click **OK** to confirm the new breakpoint properties and close the dialog box.

**Cancel**
Click **Cancel** to close the dialog box and abandon the breakpoint setting.

**Help**
Click **Help** to get online help on the controls in this dialog box.

---

**Note**
Depending on the Break/Tracepoint Type you select, the Location or the Value Match field might be unavailable. In this case, the field is grayed out.

### 4.3.2 Setting simple conditional breakpoints

The **Simple Breakpoints** menu, shown in Figure 4-7 on page 4-10, also enables you to set simple breakpoints quickly from the default Code window:

**Set from Function/Label list...**
Enables you to set a breakpoint on any number of the function names and labels in your image.

- The Function Breakpoint/Profile Selector dialog box does not provide a record of breakpoints already set, that is, when you next open this dialog box existing breakpoints are not checked.

**Simple Break if X...**
Displays the dialog box, shown in Figure 4-11 on page 4-14, enabling you to specify an expression that evaluates to an address.
Working with Breakpoints

Figure 4-11 Simple Break if X dialog box

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

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

**Simple Break if X, N times...**

This option is similar to the previous option except that now you can specify how many times execution must arrive at the specified address before the breakpoint triggers. Select this option to display a dialog box where you can specify an address for a breakpoint, shown in Figure 4-12.

![Simple Break if X, N times dialog box](image)

Figure 4-12 Simple Break if X, N times dialog box

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

--- **Note** ---

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

---

**Simple Break if X, when Y is True...**

Displays a dialog box where you can specify an address for a breakpoint, shown in Figure 4-13 on page 4-15.
Working with Breakpoints

The additional field, when expression is True, enables you to specify an expression (given in C format) that must be true when execution arrives at the specified address for the breakpoint to be triggered.

Named...

When working with a user-defined project or an auto-project, you can specify named, or standard, breakpoints that are saved in the project and so are available during your debugging session.

If enabled, click this option to select project-specific breakpoints from the predefined list, shown in Figure 4-14.

Processor Events...

If supported by your debug target, RealView Debugger maintains a list of processor events that automatically trigger a breakpoint in any application program. During your debugging session you can examine this list and select, or deselect, halting events. Select this option to display a list selection box, shown in Figure 4-15 on page 4-16.
Working with Breakpoints

The list box shows processor events that stop execution. An event is enabled when the associated check box contains a tick. Click on a check box to enable, or disable, a chosen event.

These are global breakpoints, that is they apply to processor events and not addresses.

4.3.3 Breakpoint operations

The Simple Breakpoints menu enables you to complete certain breakpoint operations:

Toggle Break at Cursor

Sets a simple breakpoint at the address defined by the position of the cursor in your code view. If a breakpoint already exists at this address use this option to clear it.

Enable/Disable Break at Cursor

Enables a breakpoint at the address defined by the position of the cursor in your code view. If there is no disabled breakpoint at this position, select this option to create a new breakpoint. If an enabled breakpoint already exists at this address select this option to disable it.

Clear All Break/Tracepoints

Clears all breakpoints set on the current target.

See Disabling and clearing breakpoints on page 4-39 for more details of disabling and clearing breakpoints.
4.3.4 Setting breakpoints from saved lists

Your personal history file, exphist.sav, is saved in your RealView Debugger home directory and is updated when you close down at the end of your session. It contains a snapshot of the current breakpoints across all your debug targets. The items in this list accumulate during this, and previous, debugging sessions.

See Saving breakpoints as favorites on page 4-41 for details of creating breakpoint favorites and adding existing breakpoints to your personal favorites list.
4.4 Setting conditional breakpoints

When setting conditional breakpoints in your application program you can specify actions and qualifiers that control how RealView Debugger handles the breakpoint:

**Qualifiers**  Test conditions that must be satisfied to trigger the breakpoint, for example testing a variable for a given value, or executing a function a set number of times, or successfully running a macro.

**Actions**  These are debugger actions completed when the breakpoint triggers, for example displaying a message or updating a window.

**Continuation state**

Specify the execution behavior immediately following completion of the actions, that is the program can stop or continue.

This section describes how to manage actions and qualifiers in conditional breakpoints:

- *Managing qualifiers and actions*
- *Attaching macros to breakpoints* on page 4-21.

4.4.1 Managing qualifiers and actions

Qualifiers, actions, and the continuation state, are set up using the Set Address/Data Break/Tracepoint dialog box, shown in Figure 4-8 on page 4-11.

**Using the New Qualifiers menu**

Click the **New** button in the Qualifiers group to display the **New Qualifiers** menu, shown in Figure 4-9 on page 4-12. You can use this to specify qualifiers when you first create a breakpoint, or to add qualifiers to edit an existing breakpoint.

This menu enables you to select the qualifier that controls execution, that is to define the condition that must be satisfied to trigger the breakpoint:

**SW Pass Count...**

Enables you to specify the number of times execution must arrive at the specified address before execution stops.

**When Expression True...**

Enables you to specify an expression that must evaluate to true to stop execution.
When Expression False...

Enables you to specify an expression that must evaluate to false to stop execution.

User Macro...

Enables you to specify a macro that runs when execution stops. This brings up a dialog box where you supply the macro name and any arguments required to run it. See Attaching macros to breakpoints on page 4-21 for an example of attaching a macro to a breakpoint.

C++ Object...

Enables you to specify a C++ this object to test. The Call Stack pane contains a This tab where you can view such objects.

Favorites...

Select this option to display your personal favorites list of breakpoint qualifiers. From here you can specify the required qualifier. See Saving breakpoints as favorites on page 4-41 for details of creating breakpoint favorites and adding qualifiers to the list.

Using the New Actions menu

Click the New button in the Actions group to display the New Actions menu, shown in Figure 4-10 on page 4-13. You can use this to specify breakpoint actions when you first create a breakpoint, or to add actions to edit an existing breakpoint. These actions are not actioned until the breakpoint qualifiers complete:

Update Window...

Displays a list selection box where you can choose which debugger windows are updated when execution stops. The list includes all windows and panes available from the default desktop. You can also redirect debugger output to a user window and this can be updated when execution stops.

When updating multiple windows, you can only use this selection box to specify one window at a time.

Update All Windows

Updates all desktop windows at the time the breakpoint triggers.

Update Sample...

Updates registered samples when execution stops. This is used as part of the graphing and visualization functions in RealView Debugger.
Note

This menu option is not available for visualization functions in this release. This option is available when a sampling variant is available, for example logging from breaks.

Breakpoint Timer

This option is enabled where your debug target supports cycle timing in hardware. The timer measures execution time from the point where the breakpoint triggers.

Message Output...

Displays a dialog box where you can enter a short text string for display when the breakpoint triggers.

By default this message appears in the Output pane but you can specify a window, for example, $250$Stop at convert proc. This sends the message Stop at convert proc to the specified window (see Attaching macros to breakpoints on page 4-21 for an example).

Favorites...

Select this option to display your personal favorites list of breakpoint actions. From here you can specify the required action.

If you have used actions previously, these are displayed at the bottom of the New Actions menu.

Viewing qualifiers and actions

When you have set up actions and qualifiers they are displayed in the Qualifiers and Actions group display lists, shown in the example in Figure 4-16 on page 4-21.
The order of the qualifiers, in the Qualifiers group display list, defines the order they are tested to trigger the breakpoint. If a qualifier fails then subsequent conditions are not tested.

The order of the actions, in the Actions group display list, defines the order they are carried out when the breakpoint triggers. These actions do not execute until all breakpoint qualifiers complete successfully.

To manage the Qualifiers and Actions display lists:

- re-order the list by highlighting a qualifier, or action, and use the up, or down, button to reposition the specified entry in the list
- highlight a qualifier, or action, and click Del to delete the specified entry
- highlight a qualifier, or action, and click Edit to update it so changing the behavior.

### 4.4.2 Attaching macros to breakpoints

RealView Debugger includes a macro facility that enables you to create macros containing complex procedures that are then executed on your host workstation. You can attach a macro to a breakpoint so that it is executed when the breakpoint triggers. The macro can return values that determine whether program execution continues or stops.
RealView Debugger recognizes several predefined macros containing commonly used functions. These macros can also be attached to breakpoints. However, if you are attaching a macro that you create yourself, for example the tutorial() macro created in *Using macros* on page 9-7, then this must be open in the debugger.

To open a macro ready to attach it to a breakpoint:

- Select **Debug → Include Commands from File...** to display the Select File to Include Commands from dialog box.
- Highlight the macro file, for example tutorial.inc.
- Click **Open** to open it into the debugger.

To attach a macro to a breakpoint:

1. Select **Debug → Simple Breakpoints → Address/Data...** to display the Set Address/Data Break/Tracepoint dialog box.
2. Enter the location of the breakpoint, for example 0x8704.
3. Click the **New** button in the Qualifiers group to display the **New Qualifiers** menu, shown in Figure 4-9 on page 4-12.
4. Select the option **User Macro...** to display the data entry prompt where you enter the macro name, shown in Figure 4-17.

![Figure 4-17 Breakpoint macro entry prompt](image)

This predefined macro displays a message box to halt execution if the **Yes** key is pressed.

5. Click **Set** to confirm your entry.
6. The Set Address/Data Break/Tracepoint dialog box now contains the macro in the Qualifiers group for the conditional breakpoint, shown in Figure 4-18 on page 4-23.
7. Click **OK** to confirm the breakpoint settings and so close the dialog box.

8. Click **Go** to execute the program and trigger the breakpoint.

--- **Note** ---

Execution-type commands are not valid within a breakpoint macro. See Chapter 9 *Working with Macros* for full details.
4.5 Setting hardware breakpoints

Use hardware breakpoints to set data breaks, or where your code is stored in ROM. The facilities available depend on the current debug target, that is both the target processor and the connection vehicle. Menu options related to hardware breakpoints are grayed out if your target cannot support them. Similarly, RealView Debugger displays a message if you select an option from a drop-down list box that is not supported by your debug target.

This section describes:

- Setting simple hardware breakpoints
- Using ranges and masks on page 4-27
- Setting hardware breakpoints on a DSP-based debug target on page 4-29.

4.5.1 Setting simple hardware breakpoints

To set hardware breakpoints, you must be connected to a debug target that supports these features, for example an ARM core with embedded ICE logic. The options shown depend on the target support for breakpoints in hardware.

Select **Debug → Simple Breakpoints → Address/Data**... to display the Set Address/Data Break/Tracepoint dialog box, shown in Figure 4-19.

![Set Address/Data Break/Tracepoint dialog box](image)

Figure 4-19 Set Address/Data Break/Tracepoint dialog box
Select a hardware Break/Tracepoint Type to display entries specific to the hardware support for breakpoints available on the current target:

**HW Instr**  Sets or modifies an instruction address breakpoint. This type of breakpoint enables you to perform hardware tests or to compare data values.

**HW Read**  Sets or modifies a read breakpoint at the specified memory location or address range. The breakpoint is triggered if the application reads from any part of the specified memory range. Where supported by your debug target, you can also add data tests.

**HW Write**  Sets or modifies a write breakpoint at the specified memory location or address range. The breakpoint is triggered if the application writes to any part of the specified memory range.

**HW Access**  Sets or modifies an access breakpoint at the specified memory location or address range. The breakpoint is triggered when a memory address is accessed. This type of breakpoint enables you to perform hardware tests or to compare data values.

**HW Data Value Read**
Sets or modifies a breakpoint that is triggered if a specified data value is read from any address, and then detected by the debug hardware on the target processor.

**HW Data Value Write**
Sets or modifies a breakpoint that is triggered if a specified data value is written to any address, and then detected by the debug hardware on the target processor.

**HW Data Value Access**
Sets or modifies a breakpoint that is triggered if a specified data value is accessed at any address, and then detected by the debug hardware on the target processor.

For each of these types, use the HW Support group to specify how the match is configured.

To set up the hardware breakpoint you must enter:

**Location:**  Specifies the memory location where the new breakpoint is set. Where supported by your target hardware, use the options from the right-arrow menu to qualify the location (see *Using ranges and masks* on page 4-27 for details).
**Value Match:**

Enter the data value that triggers the breakpoint.

If you use this with data breakpoints, this compares the data value that is read or written. When used with an instruction hardware breakpoint, this can test the value of an instruction. That is, it can be used to find an instruction in a given range.

Where supported by your target hardware, use the options from the right-arrow menu to qualify the value match (see *Using ranges and masks* on page 4-27 for details).

---

**Note**

Depending on the Break/Tracepoint Type you select, the Location or the Value Match field might be unavailable. In this case, the field is grayed out.

---

**HW Support**

Where your debug target supports breakpoint tests in hardware, they can be managed and edited using this group. If enabled, the display lists currently available tests, for example for an ARM-based target:

**DataSize**  Supports testing MAS signals from the core. This enables you to test the size of *data data bus* activity.

**Mode**  Supports testing nTrans signals from the core. This enables you to test the *data not translate* signal to differentiate access between a User mode and a privileged mode.

**Extern**  Supports hardware breakpoints that are dependent on some external condition.

The current test is shown in round brackets, for example *Ignore* or *Any*.

To change a selected test, highlight the test, for example *Match=Mode: (Ignore)*, and then click **Edit Value** to change how the test is defined, shown in Figure 4-20 on page 4-27.
4.5.2 Using ranges and masks

Where supported by your target hardware, you can qualify the location and value match entries using options available from the right-arrow menu.

Click the right-arrow at the side of the Location data field to display the **Address Range and Mask** menu, shown in Figure 4-21. Use options from this menu to specify an expression range or mask a group of instructions.

![Figure 4-21 Address Range and Mask menu](image)

Click the right-arrow at the side of the Value Match data field to display the **Value Range and Mask** menu, shown in Figure 4-21. Use options from this menu to test a range of values or mask a range of data values.

![Figure 4-22 Value Range and Mask menu](image)
Note

These menu options are only available where supported by your debug target and if you have specified an appropriate breakpoint type.

Choose from the available options to set up your breakpoint:

**Address/Value Range**

Enter the start address, or data value, for the breakpoint then click this option to specify a range, for example the address range 0x800FF..0x80A00. The separators .. are automatically inserted for you.

**Address/Value Range by Length**

Enter the start address, or data value, for the breakpoint then click this option to specify a range by length, for example the address range 0x800FF..+0x1111. The separators ..+ are automatically inserted for you.

**Address/Value Mask**

Enter the address, or data value, for the breakpoint then click this option to specify a mask. RealView Debugger inserts the mask for you, for example 0x800FF $MASK$=0xFFFF. Change this mask as required.

The mask is a bitwise-AND mask applied to the specified address, or data value, for example given the location 0x0111 and a mask 0x1001 the result is 0x0001.

The breakpoint triggers when the address, or data value, matches the given value after masking.

**NOT Address/Value Compare**

Enter the address, or data value, for the breakpoint then click this option to specify a NOT operation, for example $NOT$ 0x800FF.

Similarly, use this option to specify a range of addresses, or data values, to ignore, for example $NOT$ 0x0500..+0x0100.

**Autocomplete Range**

Enter a symbol and then click this option to compute the end-of-range address based on the symbol size. For example, if you enter a function then the autocompleted range is from the start to the end of the function. Similarly, enter a global variable to see the end-of-range address autocompleted as the variable storage address plus variable size.
**Note**

Many combinations of range, or mask, options are permitted. However, mixing range and mask generates a warning message to say that this is not permitted.

### 4.5.3 Setting hardware breakpoints on a DSP-based debug target

The options available from the Set Address/Data Break/Tracepoint dialog box change if, for example, you are using a DSP-based debug target.

If you are working with an Oak-based debug target, the HW Support group offers a single option, shown in Figure 4-23. The `HWPassCount` enables you to specify the number of times the test point is passed before the breakpoint triggers.

![Figure 4-23 HW Support group using a DSP](image)

To change the hardware pass count, click **Edit Value** to edit how the test is defined, shown in Figure 4-24.

![Figure 4-24 Changing the hardware pass count in a DSP](image)

Click **Set** to confirm the count and close the prompt box. The HW Support display list shows the new test.

Highlight the test and click **Reset** to restore the default settings.

Where supported by your target hardware, you can qualify the location and value match entries using options available from the right-arrow menu (see *Using ranges and masks* on page 4-27 for details).
4.6 Using complex breakpoints

Select Debug → Complex Breakpoints from the Code window main menu to display the Complex Breakpoints menu, shown in Figure 4-25.

![Figure 4-25 Complex Breakpoints menu](image)

Complex breakpoints use advanced hardware support on your target processor, or as implemented by your simulator software. The breakpoint types available depend on the:

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

Menu options are enabled where they are supported by your debug target. Check your hardware characteristics (see Viewing your hardware characteristics on page 4-3), and your vendor-supplied documentation, to determine the level of support for complex breakpoints.

4.6.1 Setting complex breakpoints

The Complex Breakpoints menu offers:

**HW Break if in Range...**

Displays a dialog box where you can specify a hardware breakpoint, shown in Figure 4-26.

![Figure 4-26 HW Break if in Range dialog box](image)

Use this to set, or modify, a breakpoint at the specified location. The breakpoint is triggered if the PC is within the given address range.
Select the **And if Data Value matches** check box if you also want to test a data value to trigger the breakpoint. Enter the data value to test in the data field, shown in Figure 4-26 on page 4-30.

**HW While in func/range, Break if X...**

Displays a dialog box where you can specify a complex breakpoint that uses two breakpoint units, shown in Figure 4-27.

![HW While in func/range, Break if X dialog box](image)

**Figure 4-27 HW While in func/range, Break if X dialog box**

Specify the function, or the address range, to test for the first breakpoint unit. The breakpoint triggers if the PC falls within the specified range.

Choose the type of breakpoint that you want to set for the second breakpoint unit, for example **HW Read**. You can click on the drop-down arrow to display a menu of possible breakpoint types.

Select the **And if Data Value matches** check box if you also want to test a data value to trigger the breakpoint. Enter the data value to test in the data field, shown in Figure 4-27.

--- **Note** ---

Setting a breakpoint this way displays the breakpoint as two entries in the Break/Tracepoints pane.

--- **Note** ---

**HW Break if X, then if Y...**

Displays a dialog box where you can specify a complex breakpoint that uses two breakpoint units, shown in Figure 4-28 on page 4-32.
Use this dialog box to set, or modify, a breakpoint based on two conditions being met, that is X and Y. The breakpoint is set at the specified memory location or address range depending on the values read.

Choose the type of breakpoint that you want to set for the first breakpoint unit, for example **HW Read**. You can click on the drop-down arrow to display a menu of possible breakpoint types.

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

Specify the second breakpoint unit (Y) in the same way.

--- Note ---

Setting a breakpoint this way displays the breakpoint as two entries in the Break/Tracepoints pane.

**HW Break on Data Value match...**

Displays a dialog box where you can specify a hardware breakpoint, shown in Figure 4-29.

Specify the range of data values to test for the breakpoint, for example the low value, and the high value, or use a mask. The breakpoint triggers if the PC falls within the specified range.
4.7 Using the Break/Tracepoints pane

The Break/Tracepoints pane provides a central point to manage all breakpoint operations. It enables you to:

- view all breakpoints currently set and their status (enabled or disabled)
- see the command used to create a specified breakpoint
- edit an existing breakpoint specification, for example you might want to change the location, update the qualifiers, or change debugging actions when the breakpoint triggers
- copy the attributes of an existing breakpoint to create a new breakpoint at another location
- create new breakpoints as an alternative to using the Debug menu
- access the Break/Tracepoints menu and Pane menu to manage your breakpoint operations.

The Break/Tracepoints pane also enables you to manage tracepoints during Trace and Analysis operations. See the chapter describing tracing in RealView Debugger v1.6 Extensions User Guide for details.

This section describes:

- Displaying the Break/Tracepoints pane
- Viewing breakpoints in the Break/Tracepoints pane on page 4-34
- Using the Break/Tracepoints menu on page 4-35
- Working in the Break/Tracepoints pane on page 4-37
- Using the Pane menu on page 4-37.

4.7.1 Displaying the Break/Tracepoints pane

Select View → Pane Views → Break/Tracepoints Pane from the Code window main menu to display the Break/Tracepoints pane, shown in Figure 4-30 on page 4-34.
Working with Breakpoints

Figure 4-30 Break/Tracepoints pane

If any breakpoints are set already, these are displayed in the new pane. If no breakpoints are set then the Break/Tracepoints pane is empty. As you create and set new breakpoints, the pane is automatically updated to display each new breakpoint.

The Break/Tracepoints pane also displays any tracepoints set, as in Figure 4-30. See the chapter describing tracing in *RealView Debugger v1.6 Extensions User Guide* for details on using Trace and Analysis features in this pane.

4.7.2 Viewing breakpoints in the Break/Tracepoints pane

The Break/Tracepoints pane shows entries in a tree view giving the Type and Value of each breakpoint you set, shown in the example in Figure 4-31.
Each breakpoint is identified by a:

- colored icon to show the breakpoint type (see Viewing breakpoints in your code view on page 4-7 for details)
- check box to show if the breakpoint is enabled or disabled. You can click on the check box to change the state of the chosen breakpoint or breakpoint unit.

In this example, the debug target is an ARM940T core using Multi-ICE. An area of memory has been designated ROM and so the BREAKINSTRUCTION command tries to auto-switch to a BREAKEXECUTION command if the given location is found to be in this part of memory. When the hardware breakpoint limit is reached for this debug target no further breakpoints can be set on code in this area.

In the next example, shown in Figure 4-32, different types of hardware breakpoints have been set, and disabled, during the session.

Figure 4-32 Hardware breakpoints in the Break/Tracepoints pane

4.7.3 Using the Break/Tracepoints menu

If you have at least one breakpoint set, right-click anywhere along the entry to display the Break/Tracepoints context menu, shown in Figure 4-33 on page 4-36.
Working with Breakpoints

The **Break/Tracepoints** menu includes options to edit and manage your breakpoints:

**Clear**  
Removes the breakpoint highlighted in the Break/Tracepoints pane. This is the same as double-clicking on the chosen breakpoint.

**Disable**  
This option changes the status of a breakpoint.  
Select **Disable** to disable a breakpoint you have set. This changes to **Enable** so that you can enable a breakpoint that has been disabled.  
Select, or unselect, the check box in the Break/Tracepoints pane to change the status in the same way.

**Reset PassCounters/Then-Enables**  
Resets specific qualifiers for a given breakpoint and then re-enables it ready for triggering as appropriate.

**Details...**  
Displays an information box giving details of the selected breakpoint.

**Break/Tracepoint Favorites...**  
Displays the Favorites Chooser/Editor where you can edit or delete entries, or select from your favorites to set a breakpoint.

**Show Code**  
The cursor moves to the location of the breakpoint in the appropriate code view:

- an open blue pointer marks the location in source-level view
- a filled blue pointer marks the location in disassembly-level view.

Other options might be available. For example when debugging multithreaded applications right-click on the breakpoint **Command** and use **Thread List...** to examine thread-specific breakpoints.
4.7.4 Working in the Break/Tracepoints pane

If you have at least one breakpoint set, you can use the Break/Tracepoints pane to:

- edit a breakpoint:
  1. Right-click on a breakpoint in the list to display the Break/Tracepoints menu.
  2. Select Edit Break/Tracepoint... from the context menu to display the Set Address/Data Break/Tracepoint dialog box where you can edit the breakpoint or breakpoint unit.

- copy an existing breakpoint to create a new breakpoint:
  1. Right-click on a breakpoint in the list to display the Break/Tracepoints menu.
  2. Select Copy Break/Tracepoint... from the context menu to display the Set Address/Data Break/Tracepoint dialog box, populated with the relevant information about the chosen breakpoint.
     Edit the definition to create a new breakpoint, for example change the location of the breakpoint, or add a qualifier.

4.7.5 Using the Pane menu

Click on the drop-down arrow on the Break/Tracepoints pane title bar to display the Pane menu. This menu includes options that are also available from the Debug menu. The relationship between the two menus is summarized in Table 4-1.

<table>
<thead>
<tr>
<th>Break/Tracepoints pane, Pane menu</th>
<th>Code window, Debug menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Break from Function/Label list</td>
<td>Simple Breakpoints → Set from Function/Label list...</td>
</tr>
<tr>
<td>Address/Data...</td>
<td>Simple Breakpoints → Address/Data...</td>
</tr>
<tr>
<td>Set BreakIf...</td>
<td>Simple Breakpoints</td>
</tr>
<tr>
<td>Named Break...</td>
<td>Simple Breakpoints → Named...</td>
</tr>
<tr>
<td>Processor Events...</td>
<td>Simple Breakpoints → Processor Events...</td>
</tr>
<tr>
<td>Break/Tracepoint History...</td>
<td>Simple Breakpoints → Break/Tracepoint History...</td>
</tr>
</tbody>
</table>
If you select the option Set BreakIf... from the Pane menu, a breakpoint type selection box is displayed containing all the breakpoints supported by your debug target.

---

**Note**

If you have not set breakpoints, you can access some of the options from the Pane menu by right-clicking on a blank area inside the Break/Tracepoints pane. This enables you to create new breakpoints, select from your personal favorites or history list, or to see your hardware support.
4.8 Disabling and clearing breakpoints

You can temporarily disable breakpoints. This does not delete the breakpoint but means that you can enable it quickly for re-use in your current debugging session.

If you disable a breakpoint, you can still view it in the Src or Dsm tab where it is shown as a white disc. Any accompanying downward pointing arrows are colored light gray.

When you clear a breakpoint it is removed from the breakpoint list. To remove breakpoints from your favorites list, use the Favorites Chooser/Editor dialog box, see Saving breakpoints as favorites on page 4-41 for details of how to do this.

This section describes:
- Disabling breakpoints
- Clearing breakpoints
- Clearing all breakpoints on page 4-40.

4.8.1 Disabling breakpoints

You can enable and disable breakpoints from the Code window:
- In the Src tab:
  1. Right-click in the left margin, or on the line number, of a line marked with a red disc, or on the disc itself.
  2. Select Disable Break from the context menu.
- In the Break/Tracepoints pane, select the check box to unselect it so disabling the breakpoint.
- Select Debug → Simple Breakpoints → Enable/Disable Break at Cursor from the main menu.

You can use these methods to:
- enable a disabled breakpoint, marked by a white disc
- enable, or disable, a conditional breakpoint, marked by a yellow disc.

4.8.2 Clearing breakpoints

You can clear breakpoints from the Code window in different ways:
- In the Src tab, double-click on the line number of a line marked with a red disc, or on the disc itself.
- Right-click on the breakpoint in the Break/Tracepoints pane to display the Break/Tracepoints menu. Select Clear from the menu.
Select Debug → Simple Breakpoints → Toggle Break at Cursor from the main menu.

You can use these methods to clear a conditional breakpoint, marked by a yellow disc.

### 4.8.3 Clearing all breakpoints

You can clear all the breakpoints set on a selected debug target in one operation from the Code window, either:

- Select Debug → Simple Breakpoints → Clear All Break/Tracepoints from the main menu.
- Click on the Pane menu in the Break/Tracepoints pane and select Clear All Break/Tracepoints.
4.9 Saving breakpoints as favorites

When you first start to use RealView Debugger, your personal favorites list is empty. You can create breakpoints and add them directly to this list or you can add breakpoints that you have been using in the current debugging session. This section explains the steps to do both:

- Creating a breakpoint favorite
- Saving existing breakpoints as favorites on page 4-42.

4.9.1 Creating a breakpoint favorite

RealView Debugger keeps a record of all breakpoints that you set during your debugging session as part of your history file. By default, at the end of your debugging session, these processor-specific lists are saved in the file `exphist.sav` in your RealView Debugger home directory. This file also keeps a record of your favorites, for example Break Qualifiers, Break Actions, and Break/Tracepoints.

To create a new breakpoint and add it to your favorites list:

1. Select `Debug → Simple Breakpoints → Break/Tracepoint Favorites...` to display the Favorites Chooser/Editor dialog box.

2. Click `New` to display the New/Edit Favorite dialog box, shown in Figure 4-34.

3. Click `OK` to confirm the entries and close the New/Edit Favorite dialog box.

4. Use the available controls to update the new breakpoint favorite:
   - `New` Displays the New/Edit Favorite dialog box, shown in Figure 4-34 where you can create a second breakpoint.
   - `Edit` Highlight a breakpoint in the display list and click this button to display the New/Edit Favorite dialog box already populated with the breakpoint details ready to change.
Delete Highlight a breakpoint in the display list and click this button to delete the chosen breakpoint from your favorites list.

Add to List Adds an existing breakpoint to your favorites list. See Saving existing breakpoints as favorites for details on using this button.

Set Sets the specified breakpoint on your current debug target. An error message is displayed if a breakpoint already exists at this location.

Close Closes the Favorites Chooser/Editor dialog box without setting a breakpoint on the current debug target. In this way you can build up a list of breakpoint favorites ready for your next debugging session.

Help Displays the online help for this dialog box.

4.9.2 Saving existing breakpoints as favorites

If you have already set some breakpoints, RealView Debugger lets you choose which to add to your favorites list so that they are available to re-use in future debugging sessions or with other build target configurations of your application program.

To add existing breakpoints to your favorites list:

1. Select View → Pane Views → Break/Tracepoints Pane to display the Break/Tracepoints pane.
2. Highlight a breakpoint in the display list that you want to add to your favorites list.
3. Right-click and select Break/Tracepoint Favorites... from the Break/Tracepoints menu. This displays the Favorites Chooser/Editor.
4. Click Add to List to add the specified breakpoint to your favorites list. This displays the New/Edit Favorite dialog box shown in Figure 4-35.

![Figure 4-35 Adding a new favorite](image)

The Expression field contains the breakpoint details. The Description field enables you to enter a short text description to help you to identify the breakpoint. This text is optional.

5. Click OK to confirm the breakpoint details and close the dialog box.
The Favorites Chooser/Editor dialog box is updated to show the new breakpoint in the display list. Because this breakpoint is already set, click **Close** to close the dialog box. If required, you can set another breakpoint from your favorites list before closing the dialog box.

The edited favorites list is saved to your `exphist.sav` file when you close RealView Debugger.
Working with Breakpoints
Chapter 5
Memory Mapping

This chapter describes how to manage target memory in the Process Control pane. It contains the following sections:

- About memory mapping on page 5-2
- Enabling and disabling memory mapping on page 5-4
- Setting up a memory map on page 5-5
- Viewing the memory map on page 5-7
- Editing map entries on page 5-11
- Setting top of memory and stack values on page 5-12
- Generating linker command files for non-ARM targets on page 5-13.
About memory mapping

Memory mapping is disabled by default when you first connect to your debug target, that is all memory is treated as RAM. If you are working with a suitable target you can enable memory mapping and then configure the memory using the Memory tab in the Process Control pane.

Memory mapping might be useful depending on the debug target you are using and the applications you are developing:

- If you are working with a simulator, or evaluation board, to develop your application program, using memory mapping ensures that you are not trying to load your image into memory that does not exist on the real hardware.
- If you are working with different types of memory, memory mapping enables you to load an image and specify the exact location of different sections of memory.
- If your application contains pointers or stacks, or other uses outside declared areas, it might not work correctly on the final hardware. Mapping memory as Auto makes these errors visible.
- You can declare and program Flash memory.
- Memory mapping can be used to generate linker information when developing a scatterloaded application. This is not supported by ARM-based targets.
- Memory mapping enables RealView Debugger to handle shared memory so that references are updated when modified by a different processor. This also prevents software breakpoints in shared code memory.
- If you are using a simulator that supports it, you can map memory to add wait states to obtain better cycle accuracy when profiling or measuring performance.
- Like register modeling, memory mapping provides a means for system developers to tell users about the memory configuration on boards, chips or in simulator models.

When working with memory mapping, you must be aware of the following:

- The top of memory value must be higher than the sum of the program base address and program size. If set incorrectly, the program might crash because of stack corruption or because the program overwrites its own code.
- There is no requirement that the top of memory address is at the true top of memory. A C or assembler program can use memory at higher addresses.
- If you are working with a scatterloaded application, you must define the location of stack and heap in your code.
Memory Mapping

- The default memory model for ARMulator creates memory pages as required through the whole address space, and, therefore, the ARMulator configuration file can specify a value for top of stack that is in high memory.

- When an image is loaded to a debug target, the memory map is checked to confirm that it is valid to load to the locations specified in the executable program. Memory is loaded and then read back to verify a successful load and to confirm that genuine memory is present. Memory sections defined as Auto are also updated to reflect the access type specified in the executable image.

- The memory map is used to define how memory contents are color coded when displayed in the Memory pane.
5.2 Enabling and disabling memory mapping

To enable memory mapping before you load an image:

1. Connect to a suitable target.
2. Select View → Pane Views → Memory Map to display the Process Control pane and bring the Map tab to the front.
3. Right-click anywhere in the Map tab to display the context menu.
4. Click on the option Enable Memory Mapping.
   This enables memory mapping ready to load your image.
5. To disable memory mapping, right-click anywhere in the Map tab to display the Map context menu, shown in Figure 5-1.

   ![Map context menu](image)

   Figure 5-1 Map context menu

6. Click on the option Enable Memory Mapping so that it is unselected.
   This disables memory mapping ready to load your image.

See Using the Map tab context menu on page 5-9 for details on the other options available from this menu.
5.3 Setting up a memory map

Mapping memory, before you load an image for debugging, enables you to have full access to all the memory on your debug target. You can do this:

• as part of your target configuration settings
• using an include file
• interactively using the Map tab
• by submitting the appropriate CLI commands.

In this example, you are going to set up memory manually for the current session. Target memory settings defined in this way are only temporary and are lost when you close down RealView Debugger. See the chapter describing configuring custom targets in RealView Debugger v1.6 Target Configuration Guide for details of how to configure a memory map as part of your target configuration settings.

With memory mapping enabled, set up your map in the Process Control pane:

1. Right-click on the Start entry in the Map tab to display the context menu, shown in Figure 5-1 on page 5-4.

2. Select Add or Copy Map Entry... to display the Add/Copy/Edit Memory Map dialog box, shown in Figure 5-2.

3. Change the dialog box to change the default memory mapping as follows:
   a. Enter 0x0 in the Start Addr field.
   b. Enter 0x8000 in the End field.
Memory Mapping

--- Note ---
In RealView Debugger, memory mapping is defined by a start address and a block size by default, not by an end address. Ensure that the End is inclusive Length (vs Addr.) check box is unselected to specify the end address.

4. Enter Area before image in the Description field to describe this block.
5. Click OK to confirm your changes and the Process Control Map tab is updated.
6. Set up the second block of memory using these settings:
   a. Start address = 0x8000
   b. Length = 0x8000
   c. Description = Middle
7. Click OK to confirm your changes and the Process Control Map tab is updated.
8. Set up the third block of memory using these settings:
   a. Start address = 0x10000
   b. Length = 0xFFFF0000
   c. Description = Area after image
9. Click OK to confirm your changes and the Process Control Map tab is updated, shown in Figure 5-3.

Figure 5-3 Memory mapped
5.4 Viewing the memory map

The Process Control pane provides a view of the memory mapping for the debug target that is running your application. This section describes how to use the map:

- Working with the Map tab
- Memory map configuration on page 5-8
- Using the Map tab context menu on page 5-9.

5.4.1 Working with the Map tab

To view the memory map:

1. Connect to a suitable debug target.
2. Enable memory mapping (see Enabling and disabling memory mapping on page 5-4).
3. Load an image, for example dhrystone.axf
4. Select View → Pane Views → Memory Map to display the Process Control pane and bring the Map tab to the front.
5. Click on the plus signs to expand the entries, shown in Figure 5-4 on page 5-9.

The Map tab displays a tree-like structure for each component of the memory map showing the start address, size, and access rule. A one-line text description can also be included. The way that memory is shown depends on your debug target because RealView Debugger populates this tab from:

- built-in knowledge about the processor
- target configuration information
- the description in the execution vehicle.

For example, if you are working with an ARM-based target, the first entry in the memory map shows Start (see Figure 5-4 on page 5-9) if the memory access rule is defined as Any. If you are using a DSP-based target, the first entry in the map shows the access rule for the type of memory at that location, for example Prog. Colored icons are used to show the type of memory defined, see Display colors on page 5-8.

With an image loaded, the Map tab is updated from details in the image itself. The memory map is also automatically updated if any registers change that affect memory mapping.
5.4.2 Memory map configuration

The memory map for a chosen processor is configured under the following headings:

**Type**  
The type of memory page, for example *Prog*, *I/O*, *Data*. Where no such definition is given, the type is set to *Any*.

**Access**  
Defines the access rules for the memory:
- **RAM**  Memory is readable and writable.
- **ROM**  Memory is read-only.
- **WOM**  Memory is write-only.
- **NOM**  No memory.
- **Auto**  Memory is defined by the application currently loaded. If there is no application loaded, this shows NOM.
- **Prompt**  You are prompted to confirm that this type of memory is permitted for the loaded application. If there is no application loaded, this shows NOM.
- **Flash**  Memory is readable and, if a Flash programming routine is present, writable.

**Start**  
The memory area is defined by the start address and the size. This defines the start address of the memory area.

**Size**  
 Defines the size of the memory area.

**Filled**  
This shows if a range contains data loaded from an application program.

**Description**  
A text description of the purpose of the memory supplied as part of the automatic mapping. You can also supply this information yourself (see *Setting up a memory map* on page 5-5).

To see details about a map entry, right-click on the chosen entry and select **Properties** from the context menu. This displays a text description of the type of memory defined at this location.

**Display colors**

When using the **Map** tab to view the memory map, RealView Debugger uses color to make the display easier to read and to highlight the different memory definitions, shown in the example in *Figure 5-4* on page 5-9.
In this example, memory has been mapped, using a board/chip definition file, to declare Flash. See the chapter describing configuring custom targets in *RealView Debugger v1.6 Target Configuration Guide* for details of configuring your target this way.

Colored icons enable you to identify the memory access defined:

- white (open) specifies *Any*, where no memory type is defined
- blue indicates RAM
- yellow indicates ROM
- green indicates Flash memory known to RealView Debugger
- red cross indicates no accessible memory is defined.

### 5.4.3 Using the Map tab context menu

The **Map** tab context menu, shown in Figure 5-1 on page 5-4, enables you to add new mappings or to update existing ones. The options are:

**Add or Copy Map Entry...**

Displays the Add/Copy/Edit Memory Map dialog box, shown in Figure 5-2 on page 5-5, where you can create a new map entry based on an existing entry.

**Edit Map Entry...**

Displays the Add/Copy/Edit Memory Map dialog box, shown in Figure 5-2 on page 5-5, where you can edit a map entry.
Memory Mapping

Update Map based on Image

Updates the memory map based on details held in the image.

Update Map based on Processor

Reads those registers that affect the memory maps. This is done automatically for built-in map registers but might be required if you are using external map registers, defined in the target configuration settings. Select this option to force RealView Debugger to read the registers and so update the memory map.

Save Map to Linker Command file...

Writes the current map state to a new or existing linker command file. This inserts or edits the MEMORY definitions in the linker command file, allowing for proper loading of an application based on actual memory settings. See Generating linker command files for non-ARM targets on page 5-13 for full details of how to generate this file for non-ARM targets.

Delete Map Entry

Deletes the map entry under the pointer. There is no undo.

Reset Map (Delete All)

Redefines the memory map to the initial state based on processor information, target configuration information, and processor registers. There is no undo.
5.5 Editing map entries

To edit memory map settings using the Map tab:

1. Right-click on the first entry in the display list to display the context menu shown in Figure 5-1 on page 5-4.

2. Select the option Add or Copy Map Entry... to display the Add/Copy/Edit Memory Map dialog shown in Figure 5-2 on page 5-5.

3. Use the Start Addr field to define the starting location for the mapping. This already contains the start address for the chosen block, shown in the Map tab.

4. Use the End field to define the block size for the mapping. By default, this specifies the size of the memory block to be defined. If you want to specify the end address, rather than the block size, unselect the check box End is inclusive Length (vs. Addr) and then enter the address in the End field, for example 0xFFFFFFFF0. RealView Debugger automatically sets the size you specify. If the computed size does not fall on a page boundary an error dialog is displayed and you must resubmit the block size.

   Entering a value of 0x0 remaps all memory from the starting address.

5. Highlight the access type in the display list, for example RAM.

6. Enter the memory type to be allocated, for example Any.

7. Enter a description of the new memory map settings, for example New test memory entry.

8. Click OK to confirm your new settings and to update the Map tab.

RealView Debugger displays a warning if you have entered any values incorrectly, for example a mismatch on start and end addresses. Correct these entries and click OK. When all entries are correct, the dialog box closes and the Map tab is updated.

5.5.1 Updating map entries based on registers

If you are connected to a debug target that uses register-controlled remapping, for example the ARM Integrator/AP board, the Map tab also displays the effects of any changes made to these registers. In this case, right-click on the first entry and select Update Map based on Processor from the context menu (shown in Figure 5-1 on page 5-4) to update the display based on these memory-mapped registers.
5.6 Setting top of memory and stack values

If defined, the top of memory variable specifies the highest address in memory that the C library can use for stack space. By default, a semihosting call returns stack base. Base of heap is then set to follow on directly from the end of the image data region.

You can create your own settings to specify the bottom of the stack address, the size of the stack, the bottom of the heap address, and the size of the heap. If you do not set these values manually, RealView Debugger uses default settings that are target-dependent. For ARM processors the default is 0x20000.

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

Warning: No stack/heap or top of memory defined - using defaults.

To avoid this message, set permanent values for top of memory, stack base and limit, using the Connection Properties window. Configure your debug target and define these settings so that they are used whenever you connect with RealView Debugger. See the chapter describing configuring custom targets in RealView Debugger v1.6 Target Configuration Guide for details of how memory is configured in ARM-based debug targets, and for an example of how to set up your memory map.

You can set top of memory, and other ARM-specific runtime controls, as part of a project definition. However, the available options depend on your target processor and execution vehicle. See Appendix B Project Properties Reference for details on these entries in your project SETTINGS group.

You can also set top of memory, stack, and heap values on a temporary basis, that is for the current session, using the @top_of_memory register. To do this select Debug → Memory/Register Operations → Set Register… to display the Interactive Register Setting dialog box, where the register contents can be changed.

Note
If you are using the default ARMulator to simulate an ARM processor, this is not a suitable target for setting top of memory in this way.
5.7 Generating linker command files for non-ARM targets

The memory map, shown in the Process Control Map tab, can be used to generate or modify a MEMORY section of a linker command file used in your application building. This MEMORY directive information can then be used to correctly position various sections of an application. For details of how to set up such command options see Chapter 11 Managing Projects.

To generate or modify a linker command file:

1. Right-click on the start address at the top of the entries and select Save Map to Linker Command File... from the context menu.

2. Specify the location of the file in the Select Linker Command File to Create or Modify dialog box. Remember that:
   - If the file already exists, RealView Debugger looks for a MEMORY directive block created previously and, if found, replaces that block.
   - If the file already exists, but no MEMORY directive block exists, RealView Debugger locates the first MEMORY section and inserts the MEMORY directive block before it.
   - If the file already exists, RealView Debugger makes a backup copy before updating the contents.
   - If there is no existing file, RealView Debugger creates the specified file ready to accept the MEMORY directive block.

The RealView Debugger linker command file generation process uses the built-in automatic memory mapping to generate data based on the connected target settings, for example the registers that control mapping.

The data recorded in the generated MEMORY block includes each internal RAM, ROM, and Flash section as appropriate. Each section is allocated a predefined name. All external memory added using the Map tab, or defined automatically from a loaded image, is allocated a name based on the characteristics of the memory.

The linker file format is processor-specific. If none is known, RealView Debugger uses a default format based on TI tools.

An example of a generated linker command file, in DSP format, is shown in Example 5-1 on page 5-14.
Example 5-1

/* Linker Command file for the DSPxx processor */
/* This file was generated by RVDEBUG. You can edit everything outside the MEMORY block defined by RVDEBUG. Updates by RVDEBUG will only effect that block. */
/* RVDEBUG: generated data block. Updated Fri Apr 05 15:10:41 2002
Do not modify this block. Do not put MEMORY lines above this line, put below end of this block. */
MEMORY
{
  /* Register @YYYY has (masked) value 0068 */
  PAGE 0: PDaRam: org=0x0080, len=0x177F /* internal 'Dual-Access' */
  PAGE 0: P_RAM: org=0x8000, len=0x032D /* external 'Sect .text' */
  PAGE 1: DMapReg: org=0x0000, len=0x005F /* internal 'Registers (mapped)' */
  PAGE 1: DScrDaRam: org=0x0060, len=0x001F /* internal 'Scratch Dual-Access' */
  PAGE 1: DLDaRam: org=0x0080, len=0x177F /* internal 'Dual-Access' */
  PAGE 1: D_RAM: org=0x8000, len=0x0085 /* external 'Sect .bss,.stack' */
  PAGE 1: DHRom(R): org=0xC000, len=0x3EFF /* internal 'Internal program-ROM' */
}
/* RVDEBUG: generated data above */

This example shows a combination of internal memory based on current register settings (@YYYY) as well as external memory as defined by the loading of a program.

The following notes apply to this automatic file generation process:

- If RealView Debugger creates the linker command file a comments section is inserted in the file showing that it was generated by RealView Debugger, shown in Example 5-1.

- If a file already exists and is being updated by RealView Debugger, the comments section is not inserted. RealView Debugger then inserts the generated commands above the original user-generated commands.

- If a file already exists and contains a RealView Debugger generated data block then this section is replaced when RealView Debugger updates the command file.
Chapter 6
Monitoring Execution

This chapter describes how to monitor your application program during execution using panes and views in the Code window. It contains the following sections:

- Working with registers on page 6-2
- Working with memory on page 6-12
- Working with the stack on page 6-23
- Using the call stack on page 6-28
- Working with watches on page 6-32.
6.1 Working with registers

The Register pane displays the contents of processor registers and enables you to change those contents. Where appropriate, the pane shows registers using enumerations to make it easier to read the details, and enables you to enter new values in this format. The Register pane updates the register values to correspond to the current program status each time the target processor stops.

This section describes the options available when working with registers. It contains the following sections:

- **Semihosting**
- **Displaying register contents** on page 6-3
- **Formatting options** on page 6-5
- **Changing register contents** on page 6-6
- **Managing multiple targets** on page 6-7
- **Viewing internal variables** on page 6-9
- **DCC semihosting** on page 6-10
- **Defining new registers** on page 6-11
- **Interactive operations** on page 6-11.

6.1.1 Semihosting

Semihosting enables code running on an ARM-based target to use facilities on a host computer that is running RealView Debugger. Examples of such facilities include the keyboard input, screen output, and disk I/O.

The EmbeddedICE logic in ARM cores such as the ARM7TDMI, contains a debug communications channel (DCC), that enables data to be passed between the debugger and the target using the Multi-ICE interface, without stopping the program or entering debug state.

---

**Note**

RealView Debugger does not currently support the use of channel viewers.

---

If you are using Multi-ICE to connect to a target, two modes of semihosting are supported:

- **Standard semihosting**, where the target processor enters debug state while the semihosting operation is performed.

- **DCC semihosting**, where a handler is automatically loaded to the target. Communication between the handler and the host is performed over the DCC.
DCC semihosting has two advantages over breakpoint-based semihosting:

- it is generally faster
- interrupts continue to be serviced because the target processor does not enter debug state.

**Note**

Standard semihosting is the default choice because DCC semihosting is more intrusive on the target.

For more details on DCC semihosting with RealView Debugger, see *DCC semihosting* on page 6-10.

### 6.1.2 Displaying register contents

To examine the contents of registers:

1. Connect to your target and load an image, for example dhrystone.axf.
2. Select **View** → **Pane Views** → **Registers** to display the Register pane and bring the **Core** tab to the front.
3. Click on the **Src** tab to view the source file dhrystone.c.
4. Set a simple breakpoint by double-clicking on line 150.
5. Click **Go** to start execution.
6. Enter 5000 when asked for the number of runs.
   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.
7. The contents of the Register pane are updated to show the program status as the target stops, shown in Figure 6-1 on page 6-4.
The Register pane displays tabs appropriate to the target processor running your image and the execution vehicle. Different target processors contain different registers and so the contents of this pane change depending on the target you are debugging. For example, if the target processor is a DSP, an extra tab is available, Status, displaying details of the status flag.

**Note**

If your target processor is configured using .bcd files, additional tabs are also shown, for example, if you are using an ARM Integrator/AP board, additional tabs are displayed.

8. Click **High-level Step Into** to execute one instruction and then stop. Register values that have changed, since the last update, are displayed in dark blue.

9. Click **High-level Step Into** a few more times and examine the register values as they change.

10. Right-click on a changed register and select **View Memory At Value** to use the chosen value as the starting address for a memory display.

   This displays the memory view in the last-used Memory pane. If a memory view is not visible, the default Memory pane, in the middle pane row, is used to display the view.

11. Monitor changes in the Register pane as you step through your program.

12. Double-click on the red marker disc to clear the breakpoint at line 150.
Display colors

When using the Register pane to view register contents, RealView Debugger uses color
to make the display easier to read and to highlight significant events:

- Black indicates values that are unchanged for the previous two updates.
- Dark blue shows those values that have changed since the last update.
- Light blue indicates a value that changed at the previous update.

6.1.3 Formatting options

You can change the way register values are displayed in the Register pane.

For all registers

Click the Pane menu to select formatting options for all registers currently displayed,
for example, to display contents as values rather than enumerations or to display values
in decimal format.

Use the option Copy Pane to take a snapshot of the top-level pane display. You can then
copy this, for example into a text editor, so that you can compare registers and values.

For selected registers

You can change the display format for a single register while viewing other registers in
the default format. Right-click on a chosen register, for example R3, to display the
Register context menu.

This menu enables you to change the format of the chosen register, to view the
properties, or to change the register contents, for example, select Increment to add one
to the current value. You can also select from a list of previously used values to update
the current contents.

The options offered on this menu vary depending on the register currently under the
mouse pointer. Position your pointer over the mode field of the CPSR register SVC and
then right-click to display the Status register context menu.

You can use this menu to change the register contents, or pick from a list of values
appropriate to this register.

Select Set Enumeration... to display the selection box shown in Figure 6-2 on
page 6-6.
This dialog enables you to select a new value for the chosen register or to use an expression to define the contents. For example, highlight the entry <Set by Value/Expression...>. Click OK to display a box where you can enter the value or choose from a drop-down list.

6.1.4 Changing register contents

You can use in-place editing to change register contents in the Register pane. There are, however, other ways to set register values from the Code window:

- Set the contents of a register by pasting variables from other windows. For example, right-click on a value in the source-level view and select Copy from the context menu. Right-click in the register whose value you want to change and select Paste Value from the context menu.

- Highlight a value in the Src tab in the File Editor pane and then use drag-and-drop to copy the expression into the contents of a chosen register in the Register pane.

- Select Debug → Memory/Register Operations → Set Register... from the Code window main menu to display the Interactive Register Setting dialog box.

Any register contents you change are displayed in blue.

Note
You can also set breakpoints on memory mapped registers but not on core registers, because breakpoints are set on addresses and core registers do not have addresses.

Updating register contents

The Register pane updates automatically when register contents change. This is set by default, as indicated by the checked option Automatic Update on the Pane menu.
If you change this default option, you must update the pane display manually. Select Update Window Now from the Register pane context menu to update the Register view. RealView Debugger updates the pane contents and displays changed values in blue for improved readability.

### 6.1.5 Managing multiple targets

If you are licensed to use multiprocessor debugging mode you can access different registers on multiple debug targets at the same time. To do this, set up multiple Code windows, attach each window to a different debug target and then display the registers for each target. RealView Debugger enables you to set up several Register panes with different formatting options for each.

In the first example, shown in Figure 6-3, the Register pane shows register contents for an ARM920T core (using Multi-ICE).

![Image](image.png)

**Figure 6-3 Registers for an ARM920T debug target**

In this example, you can see the CP15 tab and other tabs relating to processor-specific operations, for example Cache Operations and TLB Operations (Translation Lookaside Buffers). These are special registers and are described in full in the processor hardware documentation.

In the second example, shown in Figure 6-4 on page 6-8, the Register pane shows register contents for an ARM940T core (using Multi-ICE).
Monitoring Execution

Figure 6-4 Registers for an ARM940T debug target

Here you can see the CP15 tab and other tabs relating to processor-specific operations, for example Data Regions. These are described in full in the processor hardware documentation.

In the last example, shown in Figure 6-5, the Register pane displays register contents for an Oak-based debug target.

Figure 6-5 Registers for an Oak-based debug target

Here you can see the Status tab displaying the status registers for the debug target.

For details on setting up multiple Code windows and attaching to different debug targets, see the multiprocessing chapter in RealView Debugger v1.6 Extensions User Guide.
Monitoring Execution

6.1.6

Viewing internal variables
RealView Debugger internals are stored with the image for persistence across different
debugging sessions. These variables are displayed in the Debug tab in the Register
pane.
If you are using ARMulator to simulate a target processor, the Debug tab displays a
range of internal variables and statistics, along with selected RDI properties:
vector_catch
This variable defines which exceptions in the processor are intercepted by
RealView Debugger.
semihost_enabled
Semihosting is a mechanism for ARM targets to communicate
input/output requests from application code to a host computer running a
debugger.
Set this variable to STD (1) to enable semihosting. This is the default.
Set this variable to NO (0) to disable semihosting.
The S bit in vector_catch has no effect unless semihosting is disabled.
semihost_arm_swi
This variable defines ARM software interrupt number reserved for
semihosting.
semihost_thumb_swi
This variable defines Thumb software interrupt number reserved for
semihosting.
semihost_vector
Specific to ARM target vehicles, this variable defines the handler
address. Although this is available to any ARM target vehicle, it is only
usable if you are connecting using Multi-ICE.
semihost_window
This enables output to go to, and come from, a file or window that has
been opened with the FOPEN or VOPEN command.
irq

ARM DUI 0153C

A target can export this variable to provide a means of asserting the
interrupt request pin. To trigger an interrupt manually, set the value to 1.
To clear the interrupt, set the value to 0. To take the interrupt exception a
processor must have IRQ enabled in the CPSR.

Copyright © 2002, 2003 ARM Limited. All rights reserved.

6-9


Monitoring Execution

fiq  A target can export this variable to provide a means of asserting the fast interrupt request pin. To trigger a fast interrupt manually, set the value to 1. To clear the fast interrupt, set it to 0. To take the interrupt exception a processor must have FIQ enabled in the CPSR.

config  If you are using a debug target that contains a CP15 register, this variable contains the P, D, B, L, and Z bits. This variable is meaningless on processors without a CP15. This variable is read-only.

aci_command  If you are using ARMulator in an ARM verification system, this internal string variable passes a command to the verification kernel. See the documentation for your ARM verification system for details of the commands supported by this variable.

acmd  A string variable used to pass a command to the target processor where this is an ARM core. For internal use only.

cputime  This variable applies to ARMulator only and contains the best estimate of the time the processor has been running, measured in clock units. A clock unit is the reciprocal of the ARMulator clock speed setting. This variable is unavailable where the ARMulator clock speed is set to real-time. This variable is read-only.

sys_clock  Returns the number of centiseconds since the execution started. Values returned by this SWI might be of limited use for some benchmarking purposes because of communication overhead or other agent-specific factors.

These variables depend on your debug target. See your processor hardware documentation for details on processor-specific statistics.

6.1.7 DCC semihosting

If you are using Multi-ICE to connect to a target, the Debug tab displays other variables, shown in Figure 6-6 on page 6-11.
Specify DCC semihosting by setting the variable `semihost_enabled` to DCC, as shown in Figure 6-6. This means that the DCC semihosting software interrupt handler is installed in memory at the address specified by the `semihost_dcchndlr_addr` variable. It is essential that a region of memory starting at this address is available in target memory and is unused. The default address stored in this variable is 0x70000. You might have to change this to a lower value to suit the target memory.

For full details on DCC semihosting with Multi-ICE, see the Multi-ICE User Guide.

### 6.1.8 Defining new registers

RealView Debugger has built-in awareness of core registers and other standard registers for different processor families. These are displayed in the Register pane. However, you can define new ASIC registers using the Advanced_Information blocks in your target configuration settings. When configured, user-defined registers can be displayed in the Register pane in the same way as standard registers. See the chapter describing configuring custom targets in RealView Debugger v1.6 Target Configuration Guide for details.

### 6.1.9 Interactive operations

For full details on using interactive operations on register contents using the Debug menu, see Chapter 7 Reading and Writing Memory, Registers, and Flash.
6.2  Working with memory

The Memory pane displays the contents of memory and enables you to change those contents. On first opening, the pane is empty, because no starting address has been specified. If a starting address is entered, values are updated to correspond to the current program status each time your program stops.

This section describes the options available when working with memory displays and contains the following sections:

- Displaying memory contents
- Formatting options on page 6-13
- Operating on memory contents on page 6-17
- Changing memory contents on page 6-20
- Managing multiple targets on page 6-21
- Interactive operations on page 6-22.

6.2.1  Displaying memory contents

To examine the contents of memory:

1. Select **File → Reload Image to Target** to reload the image `dhrystone.axf`. You can also reload an image using the **Reload Image** button on the Actions toolbar.

2. Click on the **Src** tab to view the source file `dhry_1.c`.

3. Set a simple breakpoint by double-clicking on line 150.

4. Click **Go** to start execution.

5. Enter `5000` when asked for the number of runs.

   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.

6. Select **View → Pane Views → Memory** to display the Memory pane. Start addresses can be set using in-place editing or using the context menu.

7. Right-click in the first address in the window to display the **Address** context menu.

8. Select **Set New Start Address...** to display the selection box shown in Figure 6-7 on page 6-13.
You specify the start address by giving an address in hexadecimal or by giving a C/C++ expression that RealView Debugger computes to obtain the starting address. You can use any valid expression using constants and symbols.

You can also use the drop-down arrow to select an expression from a browser or to re-use a value entered previously. The drop-down also gives access to your list of personal favorites where you can store a memory address for re-use in this, or future, debugging sessions.

In this example, memory addresses of interest are in the region of 000088A0 so set the start address to examine memory from this location.

Numbers entered here must start with a zero. This means that RealView Debugger can distinguish these entries from valid variable names.

9. Enter the required location, for example 0x00088A4, and then click Set to update the Memory pane.

The memory display is arranged in columns. The left-most column shows the memory address. The memory contents are shown in the other columns. The number of columns displayed varies depending on the size of the pane. Color coding is used to distinguish the type of memory being displayed, see Display colors on page 6-17 for details.

10. Monitor changes in the memory display as you step through your program.

11. Double-click on the red marker disc to clear the breakpoint at line 150.

6.2.2 Formatting options

You can change the way memory contents are displayed, and set the start address, using the Pane menu:

Copy

If you have selected memory contents, use this option to copy the values to the clipboard ready to paste.
Update Window Now

If you have unselected the option Automatic Update, you can use this option to update the memory display manually. You can update the display using this option at any time when execution is stopped. This enables you to catch any memory updates made externally.

Recompute Expression Now

Where you have used a C/C++ expression to compute the start address, select this option to recompute the expression and, where necessary, start at the new location. Where you have used a fixed value to specify the start address, select this option to update only the pane contents.

Set New Start Address...

Select this option to enter a C/C++ expression to compute the start address. This displays the selection box shown in Figure 6-7 on page 6-13 with the current expression already displayed. Change the address and click Set to specify the start address for the memory view.

Previous Start Address

Uses a previous start address for displaying the contents of memory. The history list holds up to 16 previous start addresses added when:

- you enter a new start address or expression
- the current expression is recomputed to generate a new start address
- the start address is set from the Address context menu.

Set Number of Columns to show...

When the Memory pane is first opened, the number of columns you can see depends on the size of the pane and is chosen so as to show an even number of bytes. You can use this option to change the number of columns visible in the display. Use the selection box to show up to 32 columns in a single window. This number does not include the column used when the ASCII display option is selected, see Show ASCII below. The default setting is 0 which configures the number of columns to fit the window size.

Automatic Update

Updates the memory display automatically, that is when:

- you change memory from anywhere in RealView Debugger
- program execution stops.

This is the default.
Recompute Expression on Update

Where you have used a C/C++ expression to compute the start address, select this option to recompute the expression when the pane contents are updated, see above, and start at the new location where necessary.

Timed Update when Running

The memory display can be updated at a specified time interval during program execution. Select this option to set this timer according to the update period specified below.

This is only available where supported by the underlying debug target.

Timed Update Period

Use this to choose the interval, in seconds, between window updates.

Any value you enter here is only used when the option Timed Update when Running is enabled, that is where supported by the underlying debug target.

Signed Decimal

Displays the memory contents as negative or positive values where the maximum absolute value is half the maximum unsigned decimal value.

Unsigned Decimal

Displays the memory contents from 0 up to the highest value that can be stored in the number of bits available.

Hexadecimal

This displays memory contents as hexadecimal numbers.

Hex, leading Zeroes

Displays memory values in hexadecimal format including leading zeroes.

This is the default display format for data values in this pane.

Show ASCII

Adds another column to the Memory pane, on the right hand side, to show the ASCII value of the memory contents.

ASCII format displays column values as characters. The ASCII format is useful if, for example, you are examining the copying of strings and character arrays by transfer in and out of registers.

Any non-printable value is represented by a period (.)
Data formats

The Pane menu contains an extended panel to define how data values are displayed in the Memory pane. The display format used for viewing memory contents varies depending on the data types supported by your target processor:

Minimum Access Size

Displays memory contents in the format specified as the minimum memory access size for the target. This is the default.

Bytes (8 bits) Each column displays 8 bits of data.

Half Words (16 bits)

Each column displays 16 bits of data. Where your debug target is an ARM processor halfwords are aligned on 2-byte boundaries.

Long Words (32 bits)

Each column displays 32 bits of data. Where your debug target is an ARM processor long words are aligned on 4-byte boundaries.

Long Long Words (64 bits)

Each column displays 64 bits of data.

Fixed (word size)

Enables you to use fixed point format for displaying numeric values, that is based on the natural size for the debug target processor. The default format is unsigned and one less than the number of bits in the value.

Fixed...

Displays a selection box that enables you to specify a fixed point format to display numeric values. The value entered here becomes the default display format for the pane.

Floats (32 bits)

Displays values in floating point IEEE format, occupying four bytes, for example:

2.5579302e-041

Doubles (64 bits)

Displays values in floating point IEEE format, occupying eight bytes, for example:

4.71983561663e+164
Display colors

When using the Memory pane to view memory contents, RealView Debugger uses color to make the display easier to read and to highlight significant events:

- Black specifies RAM or memory that can be modified.
- Blue shows those contents that have changed since the last update. Light blue indicates a previous update.
- Yellow indicates the contents of ROM.
- Green indicates Flash memory known to RealView Debugger. Otherwise the values are displayed in yellow, indicating ROM.
- Red**** indicates one of:
  — no memory is defined at this location
  — memory at this location is defined as Auto meaning it is determined when loading your application program
  — memory is defined as prompt meaning that you are prompted to confirm the usage when loading the application.
- Red!!!! indicates that there has been a failure in performing the memory operation. Double-click, with the right mouse button, at this location to get an explanation of the problem.

6.2.3 Operating on memory contents

You can perform many different operations on the memory displayed in the Memory pane using the context menus. The menu shown, and the options available, depend on the type of memory under the cursor when you right-click and on the valid licenses that you have. The color-coded display helps you to identify the memory type.

Note

If you right-click on a memory cell to access a context menu, the change is made to the cell under the cursor. This is independent of any highlighted cells in the view.

Right-click on a memory address to display the Address context menu that provides options to set the start address:

Update (double right-click)

Updates the display in the Memory pane.
Set New Start Address...

Enables you to specify the starting address for the display of memory contents.

Previous

Enables you to use the previous starting address for the display of memory contents.

Recompute Expression

Where you have used a C expression to compute the start address, select this option to recompute the expression and, where necessary, starts at the new location.

Right-click on a memory cell, or byte, that is black or green, that is where the type is ROM, Flash, or modifiable, to display the menu shown in Figure 6-8.

```
Figure 6-8 Memory value (RAM) context menu
```

This menu contains the options:

**Update (double right-click)**

Updates the display in the Memory pane.

**Set Start Address from Content**

Enables you to use the cell contents as the starting address for the display of memory contents. This option is enabled when the cell contains a scalar the size of an address or pointer.

**Show Symbol from Content**

RealView Debugger looks up the address held in the cell and displays any symbol at that address.
Show Symbol at Address
Displays any symbol at the address contained in the cell, and not the contents.

Set to 0
Enables you to set the current memory cell to zero.

Increment
Enables you to add 1 to the contents of the memory cell.

Decrement
Enables you to subtract 1 from the contents of the memory cell.

Set Value...
Displays a prompt where you can enter a new value for the memory cell. This new value is then entered and the memory display is updated. A memory cell can also be changed using in-place editing.

Set Memory Interactive...
Enables you to use memory interactive operations available in RealView Debugger.

Fill Memory with Pattern...
Enables you to fill memory starting at this location.

Set Break At...
Displays the Set Address/Data Break/Tracepoint dialog box where you can specify a breakpoint on the current memory cell. The type of breakpoint offered depends on the type of memory at the chosen location. For example, if the memory is defined as ROM, RealView Debugger offers a hardware breakpoint first.

Set Trace Point...
Enables you to set tracepoints based on the current memory view.

Set Trace Range
Enables you to set tracepoints based on a range of values in the current memory view.

Right-click on a memory cell, or byte, that is yellow, that is where the type is ROM, to see the ROM-specific context menu that offers a subset of these options.
Memory errors

Where a memory cell contains !!!!(colored red), this shows that there has been an error in the memory operation. Right-click to display a menu with a single option:

Show Error Code...

Select this to display the error code returned from the debug target when the memory operation failed.

6.2.4 Changing memory contents

RealView Debugger enables you to change memory values in several different ways. For an example of how to change memory contents:

1. Select File → Reload Image to Target to reload the image dhrystone.axf.
2. Click on the Src tab to view the source file dhry_1.c.
3. Set a simple breakpoint by double-clicking on line 150.
4. Click Go to start execution.
5. Enter 5000 when asked for the number of runs.

The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.
6. Use the Pane menu, in the Memory pane, to set the start address to 0x8B00 and display the ASCII values, shown in Figure 6-9.

Figure 6-9 Example memory display

The memory locations 00008B00-00008B0E contain the four hexadecimal values 0x32, 0x27, 0x4E, and 0x44 corresponding to the string ’2’ND’.

Right-click in the value at 00008B00, that is 0x32, to display the context menu shown in Figure 6-8 on page 6-18.
This enables you to change the contents at the specified location. Select the option **Set Value...** from the context menu to display the selection box, shown in Figure 6-7 on page 6-13, where you can enter the new value.

You can also use the drop-down arrow to select from a browser or to re-use a value entered previously. The drop-down also gives access to your list of personal favorites where you can store a data value for re-use in this, or future, debugging sessions.

Enter the required hexadecimal value 0x4E, or enter ‘N’, and click **Set** to update the memory display. You can use uppercase or lowercase to enter the new value.

The new value is displayed in blue and the ASCII value changes from 2 to N.

7. Change the value at location 00008B0C from 0x27 to 0x6F (lowercase o).

8. Data values can be entered in a format that is different from the display format. Right-click in the location 00008B05 (0x4E), for example, enter the decimal value 32 and then click **Set**. The memory pane is updated with the new value, a space, displayed in the chosen display format.

9. You can also use in-place editing to change memory contents. Double-click in the value 0x44 (D) at location 00008B0E and change it to 0x32. Press Enter to confirm the new value.

   If you press Escape then any changes you made in the highlighted field are ignored.

10. View the changed values in the memory display. Each new value is displayed in blue as the pane contents are updated.

11. View the changes you have made in the messages displayed when your program completes. The string “2’ND” has been replaced by “No 2”.

12. Double-click on the red marker disc to clear the breakpoint at line 150.

### 6.2.5 Managing multiple targets

If you are licensed to use multiprocessor debugging mode you can examine different memory views on multiple debug targets at the same time. To do this, set up multiple Code windows, attach each window to a different debug target and then display the memory contents for each target. RealView Debugger enables you to set up several Memory panes with different formatting options for each.
6.2.6 Interactive operations

You can also perform operations on memory contents including saving memory to a file, and reloading, and filling memory. See Chapter 7 Reading and Writing Memory, Registers, and Flash for details.
6.3 Working with the stack

A stack, or run-time stack, is an area of memory used to store function return information and local variables.

Executing a function sets up the stack. As the new function is called, a record is created on the stack including traceback details, and local variables. At this point these arguments and local variables become available to RealView Debugger and can be accessed through the panes and windows of the Code window.

When the function returns the area of the stack occupied by that function is recovered automatically and can then be used for the next function call.

In a typical memory-managed ARM processor, the memory model comprises:

- a large area of application memory starting at the lowest address (code and static data)
- an area of memory used to satisfy program requests, the heap, that grows upwards from the top of the application space
- a dynamic area of memory for the stack which grows downwards from the top of memory.

The Stack Pointer (SP) points to the bottom of the stack.

--- Note ---

Modifying a value in the stack might cause the application program to perform incorrectly or even to abort operation completely.

RealView Debugger can provide the calling sequence of any functions that are still in the execution path because their calling addresses are still on the stack. However, when the function is off the stack, it is lost to RealView Debugger. Similarly, if the stack contains a function for which there is no debug information, RealView Debugger might not be able to trace back past it.

This section describes ways of working with the stack:

- Using the Stack pane on page 6-24
- Formatting options on page 6-25
- Operating on stack contents on page 6-26
- Context controls on page 6-26
- Setting a breakpoint on page 6-26
- Interactive operations on page 6-27.
6.3.1 Using the Stack pane

The Stack pane enables you to monitor the contents of the stack as raw memory, and to make changes to those settings. This might be especially useful for assembly language programmers. The Stack pane shows the contents of the stack at the SP register which is always kept at the top-left of the display area. Use this pane to view changes as they happen in the stack.

The Stack pane enables you to follow the flow of your application through the hierarchical structure by displaying the current state of the stack. This shows you the path that leads from the main entry point to the currently executing function.

To view the Stack pane:

1. Select File → Reload Image to Target to reload the image dhrystone.axf.
2. Select View → Pane Views → Stack to view the Stack pane.
3. Click on the Src tab to view the source file dhry_1.c.
4. Set a simple breakpoint by double-clicking on line 150.
5. Click Go to start execution.
6. Enter 5000 when asked for the number of runs.
   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.
7. View the updated Stack pane, shown in Figure 6-10.

![Figure 6-10 Viewing the stack](image)

The stack pointer, marked by SP, is located at the bottom of the stack. The frame pointer, marked by FP, shows the starting point for the storage of local variables.
8. Monitor changes in the Stack pane as you step through your program, for example by clicking **Hi-level Step Into**.

9. Double-click on the red marker disc to clear the breakpoint at line 150.

The stack is displayed in columns:

- **Address**: The left column contains the memory addresses of the stack.
  - In some target processors that use a Harvard architecture, for example a DSP, a D is appended to show that this a data address. You must include this letter when specifying such an address as the starting address.

- **Value**: The right column displays the contents of the addresses in the stack.

As with the Memory pane, the memory display in the Stack pane is color-coded for easy viewing and to enable you to monitor changes.

### 6.3.2 Formatting options

You can change the way stack contents are displayed, and set the start address, using the Pane menu. This menu provides options enabling you to manage stack contents display during your debugging session, to change the display format, and to extract data from the pane for use in other panes or windows. Highlight an entry in the display and then choose from the list of available options:

- **Copy**: Copies the chosen entry from the list to the clipboard.

- **Previous Start Address**: Enables you to use the previous starting address for the display of memory contents.

- **Signed Decimal**: Displays the memory contents as negative or positive values where the maximum absolute value is half the maximum unsigned decimal value.

- **Unsigned Decimal**: Displays the memory contents from 0 up to the highest value that can be stored in the number of bits available.

- **Hexadecimal**: Displays memory contents as hexadecimal numbers.

- **Hex, leading Zeros**: Displays memory values in hexadecimal format including leading zeroes. This is the default display format for data values in this pane.
Show ASCII Adds another column to the Stack pane, on the right hand side, to show the ASCII value of the memory contents.

ASCII format displays column values as characters. The ASCII format is useful if, for example, you are examining the copying of strings and character arrays by transfer in and out of registers.

Any non-printable value is represented by a period (.)

6.3.3 Operating on stack contents

You can perform operations on the memory displayed in the Stack pane using the memory contents context menus, as described in "Operating on memory contents" on page 6-17.

6.3.4 Context controls

There are two Context controls available from the Code window main menu:

Stack up Moves up one stack level from the current scope location giving access to all local variables at that location. A stack level is determined by each calling function.

Stack down Moves down one stack level from the current scope location giving access to all local variables at that location. A stack level is determined by each calling function.

You must use the Stack up control first, because the context is as far down the stack as possible.

6.3.5 Setting a breakpoint

To set a breakpoint in the Stack pane:

1. Right-click on the required value.
2. Select Set Break At... from the Stack Value context menu.
3. Complete the entries in the Set Address/Data Break/Tracepoint dialog box.
4. Click OK to close the dialog box and set the breakpoint.

RealView Debugger sets a breakpoint on a symbol address where it exists on the stack. As soon as you exit the function, the address is no longer meaningful. Do not, therefore, use such a breakpoint where execution runs past the function return call.

Unlike the Watch pane, the Stack pane acts like a snapshot of a chosen address. It does not track each invocation of a function and so is not able to track the chosen symbol.
6.3.6 Interactive operations

You can also perform other operations on memory contents using the Stack pane. See Chapter 7 Reading and Writing Memory, Registers, and Flash for details.
6.4 Using the call stack

Processors maintain a call stack for each processor in your debug target. If you are debugging multithreaded applications, a thread stack is also maintained.

As a program function is called it is added to the stack. Similarly, as a function completes execution and returns control normally, it is removed from the stack. The stack, therefore, contains details of all functions that have been called but have not yet completed execution.

RealView Debugger includes features enabling you to monitor variables and access traceback as your debugging session develops:

- Using the Stack pane
- Using the Call Stack pane.

6.4.1 Using the Stack pane

The Stack pane enables you to monitor activity on the stack during program execution by giving access to the stack as raw memory. See Working with the stack on page 6-23 for full details.

6.4.2 Using the Call Stack pane

Use the Call Stack pane to follow the flow of your application through the hierarchical structure by examining the current status of functions and variables. This enables you to see the path that leads from the main entry point to the currently executing function at the top of the stack.

The Call Stack pane shows the:

- name of the function
- line number in the source file from which the function was called
- parameters to the function.

To use traceback:

1. Select File → Reload Image to Target to reload the image dhrystone.axf.
2. Click on the Src tab to view the source file dhry_1.c.
3. Set a simple breakpoint by double-clicking on line 150.
4. Click Go to start execution.
5. Enter 5000 when asked for the number of runs.
The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.

6. Select View → Pane Views → Call Stack to view the Call Stack pane, shown in Figure 6-11.
   When you open the Call Stack pane, the first entry is 0x0000AFFC showing the address of the link register when the application starts. This is the default format for all variables in this pane.

7. Continue to step through your program. If the current line is a multistatement line, for example a For statement, then the Call Stack pane shows the information in the form of line and column details, shown in Figure 6-11.

8. Monitor changes in the Call Stack pane as you step through your program.

9. Double-click on the red marker disc to clear the breakpoint at line 150.

Tabs in the Call Stack pane

The Call Stack pane contains the tabs:

**Call Stack**
Displays details of the functions currently on the stack, shown in Figure 6-11.

**Locals**
Displays a list of the variables that are local to the current function.

**Statics**
Displays a list of static variables local to the current module.

**This**
In C++ the this pointer locates the object for which the member function was called. It is C++ specific.

Using the Pane menu

Click on the Pane menu button to display the Pane menu that contains options to:

- manage variables
- change the display format
• change how contents are updated
• extract data from the Call Stack pane for use in other panes or windows.

—— Note ————
The options accessible from the Pane menu depend on the tab currently displayed.

Click on the Call Stack tab, highlight an entry in the functions list and display the Pane menu options:

Copy       Copy the chosen entry, where enabled.

Timed Update when Running
The Call Stack pane can be updated at a specified time interval during program execution. Select this option to set this timer according to the update period specified below.
This is only available where supported by the underlying debug target.

Timed Update Period
Use this to choose the interval, in seconds, between window updates.
Any value you enter here is only used when the option Timed Update when Running is enabled, that is where supported by the underlying debug target.

Update Window Now
Updates the contents of the Call Stack pane. Use this when Timed Update when Running is enabled, or when Automatic Update is disabled.

Automatic Update
Refreshes the Call Stack pane as soon as a watchpoint is triggered and execution stops. This is enabled by default.

Show char* and char[] as strings
Displays local variables of type char* and char[] (or casted) as strings.
This is enabled by default.

Show integers in hex
Displays all integers in hexadecimal format. Disabling this option displays all integers in decimal. This is enabled by default.

Properties    Displays a text description of the item under the cursor.
6.4.3 Using context menus

The Call Stack pane contains several context menus depending on which entry is selected and the type of variable under the cursor.

For example, right-click a function in the Call Stack tab to display the Function context menu:

Scope-to Scopes to the chosen function.

Break at... Sets a breakpoint at the address defined by the chosen function, if this is permitted.

BreakIf... Displays the Breakpoint type selection box.

Go to This continues execution until the specified point in the stack is reached.

Properties... Displays a text description of the item under the cursor.

6.4.4 Stack controls

When working with the Call Stack pane there are two Context controls available from the Actions toolbar:

- Stack up
- Stack down.

See Context controls on page 6-26 for full details.
6.5  Working with watches

Use watches to monitor variables or to evaluate expressions during your debugging session:

- Setting watches in source-level view
- Working with the Watch pane
- Managing watches on page 6-35
- Saving watches as favorites on page 6-38.

6.5.1 Setting watches in source-level view

To set a watch:

1. Select File → Reload Image to Target to reload the image dhrystone.axf.
2. Click on the Src tab to view the source file dhry_1.c.
3. Right-click on a variable name, for example Run_Index at line 146, and select Watch from the Source Variable Name menu.
4. Set a second watch, for example on the variable Enum_Loc at line 155.

If you are working in source-level view and the Watch pane is visible, you can set watches in other ways:
- use the option Enter New Expression... from the Pane menu
- drag-and-drop the variable to watch
- select a variable to watch and copy it into the Watch pane.

6.5.2 Working with the Watch pane

The Watch pane enables you to view expressions and their current values. You can use the Watch pane to create breakpoints, or to change existing watched values.

Displaying the pane

Select View → Pane Views → Watch to view the Watch pane, shown in Figure 6-12 on page 6-34.

The Watch pane contains a series of tabs. The Watch1 tab is selected by default. You can set expressions on different tabs to make it easier to manage what is being watched during your debugging session. Click on the required tab to select it.
Expressions are listed in the order they were created. You can drag the column headings to display the full name or value if required. Where an entry contains subentries, for example an array, a plus sign is appended to the name. Click on this to expand the display.

**Using the Pane menu**

Highlight an entry in the watched variables list and click the Pane menu button to display the Pane menu. This contains:

- **Cut** Select this option to copy the chosen value to the clipboard and then delete it. Where the entry has subentries, for example an array, select this option to delete all subentries.

- **Copy** Select this option to copy the chosen value from the list to the clipboard. From here it can be copied into another tab in the Watch pane, or into another pane or window.

- **Paste** Pastes the contents of the clipboard into the chosen entry. This might be an entry from another tab in the Watch pane, or text from the File Editor pane or another application.

- **Delete** If you are using in-place editing, select this option to delete the chosen value, or character, from the entry without copying it to the clipboard. You can also delete a value by highlighting it and pressing the Delete key.

**Timed Update when Running**

Enables a timed update of the Watch pane when the target processor is running. Select this option to refresh the expressions list automatically where it contains memory based expressions. Other expression types are not updated.

This is only available where supported by the underlying debug target.

**Timed Update Period**

Use this to choose the interval, in seconds, between window updates.

Any value you enter here is only used when the option **Timed Update when Running** is enabled, that is where supported by the underlying debug target.

**Update Window Now**

Updates the contents of the Watch pane. Use this when **Timed Update when Running** and **Automatic Update** have been disabled. Select this to update the current tab.
Monitoring Execution

Enter New Expression...
Displays a prompt box where you can enter the expression to be watched. This displays the name and current value in the current tab. Click the required tab before selecting this option.

Automatic Update
Refreshes the Watch pane as soon as execution stops. This is enabled by default.

Recompute Expression with same Context
If you watch an expression, the result is evaluated based on the current context. Select this option to recompute expressions using the context when set.

Show char* and char[] as strings
Displays values as strings where appropriate. This is enabled by default.

Show integers in hex
Displays all integers in hexadecimal format. Disabling this option displays all integers in decimal. This is enabled by default.

Properties
Displays a text description of the item under the cursor.

Viewing watches
As you set watches, expressions are added to the Watch pane, shown in Figure 6-12.

The entries correspond to the watches you set, in the order that you set them. Each expression is shown giving the Name and Value. You can expand the column headings by dragging on the boundary marker to make the details easier to read.

To delete an expression, highlight it and press Delete. There is no undo for this operation but saving the watches in your personal favorites list enables you to reinstate any deleted entry.
6.5.3 Managing watches

With a watch set, you might want to change the expression, change the display format used, or edit it directly to control how it is monitored. Using the Watch pane enables you to carry out these operations and gives access to context menus where editing options are available.

Figure 6-13 shows an example Watch pane, with several expressions already set.

One watched value is an array as shown by the plus sign appended to the variable name. Click on the plus sign to expand the view and display the array elements.

--- Note ---

If the chosen array is very large, RealView Debugger warns before expanding the view.

---

You can access context menus, inside the Watch pane, to control watch options and edit watches directly, see:

- Using the Name menu
- Using the Value menu on page 6-36
- Using the pane context menu on page 6-37
- Editing watches on page 6-37.

Using the Name menu

If you have set up expressions, right-click on a chosen entry Name to display the Name menu. This context menu provides options that operate on selected entries only:

**Update** Updates the displayed value for the chosen expression. This is applied in combination with any update options you set from the Pane menu.
Format... Displays a selection box to specify the format for the watched expression. Highlight the required format from the list of available formats. You can also cast top level expressions, including casting to array, for example char* and char[12].
Click OK to confirm your choice. This closes the selection box and the new format is applied to the chosen expression.

Break at... If enabled, select this option to set a breakpoint at this location.

BreakIf... If enabled, select this option to set a conditional breakpoint.

Add from Favorites... Displays the Favorites Chooser/Editor dialog box where data values saved in your personal favorites can be added to the Watch pane. See Saving watches as favorites on page 6-38 for details.

View Memory At Value Select this option to use the chosen value as the starting address for a memory display.
This displays the memory view in the last-used Memory pane if visible. If a suitable pane is not visible, the default pane in the Middle pane row is used.

View Memory At Address Select this option to use the address of the chosen item as the starting address for a memory display.
This displays the memory view in the last-used Memory pane if visible. If a suitable pane is not visible, the default pane in the Middle pane row is used.

Properties... Displays a text description of the item under the cursor.

Using the Value menu
With a watch set, you can right-click on a chosen entry Value to display the Value menu. This context menu provides options that operate on selected watches only:

Update Updates the displayed value for the chosen expression. This is applied in combination with any update options you set from the Pane menu.

Format... Displays a selection box where you can highlight the required format for the expression from the list of available formats.
Set to 0  Sets the value of the chosen expression to zero, where allowed. An error message is displayed if you try to set a value to zero when not permitted.

Increment  Adds 1 to the value of the chosen expression. An error message is displayed if you try to increment a value when not permitted.

Decrement  Subtracts 1 from the value of the chosen expression. An error message is displayed if you try to decrement a value when not permitted.

Set from Favorites...
  Displays the Favorites Chooser/Editor dialog box where data values saved in your personal favorites list can be inserted into the specified location.

Recent expressions
  The rest of this menu contains a list of recently-used variables and data values. You can re-use entries from this list as required.

Using the pane context menu

If you right-click anywhere inside an empty entry in the Watch pane, a short context menu is displayed. This provides the options:

Update All  Updates the details for all the expressions currently displayed in the Watch pane.

Add from Favorites...
  Displays the Favorites Chooser/Editor dialog box where expressions saved in your personal favorites list can be added to the Watch pane.

Editing watches

You can use in-place editing to change expressions in the Watch pane, and to add new ones:

1. Double-click in the name you want to change, or press Enter if the item is already selected. The name is enclosed in a box with the characters highlighted to show they are selected (pending deletion).
2. Enter the new name, or move the cursor to change the existing expression, or add a cast.
3. Press Enter to store the new name.

If you press Escape then any changes you made in the highlighted field are ignored.
6.5.4 Saving watches as favorites

When you first run RealView Debugger after installation, all favorites lists, stored in your `exphist.sav` file, are empty. You can create watches and then add them directly to this list or you can add watches that you have been using in the current debugging session. This section explains the steps to follow to do both.

**Creating a watch favorite**

To create a Watch favorite, right-click inside a blank entry of the Watch pane to display the Name menu, and then select Add from Favorites... This displays the Favorites Chooser/Editor dialog box. If this is the first time you have used watches in RealView Debugger, the display list is empty.

To create a new watch and add it to your favorites list:

1. Click New to display the New/Edit Favorite dialog box shown in Figure 6-14.

   ![Figure 6-14 New/Edit Favorite dialog box](image)

2. Enter the expression to be watched, for example `Ptr_Comp`.

3. Enter a short text description to help you to identify the watch for future use, for example my watch favorite.
   
   This is optional.

4. Click OK to confirm the entries and close the New/Edit Favorite dialog box.
   
   The Favorites Chooser/Editor dialog box is displayed with the newly-created watch shown in the display list.

   Duplicate entries are not permitted in the favorites list.

The Favorites Chooser/Editor dialog box contains the controls:

**New**

Displays the New/Edit Favorite dialog box shown in Figure 6-14 where you can create a second watch.

**Edit**

Highlight a watch in the display list and select this option to display the New/Edit Favorite dialog box already populated with the watch details ready for editing.
Delete  Highlight a watch in the display list and select this option to delete the chosen watch from your favorites list.

Add to List  Adds an existing watch to your favorites list. See Saving existing watches as favorites for details on using this button.

Set  Sets the specified watch on your current debug target.

Close  Closes the Favorites Chooser/Editor dialog box without setting a watch, or changing the displayed list.

Help  Displays the online help for this dialog box.

Saving existing watches as favorites

With several watches already set, RealView Debugger lets you choose which to add to your favorites list so that they are available for re-use in future debugging sessions or with other target configurations of your application program.

To add existing watches to your favorites list:

1. Highlight an expression in the Watch pane that you want to add to your favorites list.
2. Right-click on the Name and select Add from Favorites... from the Name menu. This displays the Favorites Chooser/Editor, shown in Figure 6-15.
3. Click Add to List to add the specified expression to your favorites list. This displays the New/Edit Favorite dialog box shown in Figure 6-16 on page 6-40.

Figure 6-15 Existing watches in the Favorites Chooser/Editor

The display list shows any watches already saved in your favorites list. The data field now shows the chosen expression.

Click Add to List to add the specified expression to your favorites list. This displays the New/Edit Favorite dialog box shown in Figure 6-16 on page 6-40.
The Expression field contains the chosen watch and you can enter a short text description to help you identify the watch favorite. This is optional.

4. Click **OK** to confirm the watch details and close the dialog box.

The Favorites Chooser/Editor dialog box is displayed showing the new watch in the display list. Because this watch is already set, click **Close** to close the dialog box. If required, set another watch from your favorites list before closing the dialog box.

The edited favorites list is saved to your `exphist.sav` file when you close RealView Debugger.

**Saving data values as favorites**

With several watches already set, you can right-click on the `Value` for a specified expression and display the `Value` menu. This context menu includes the option **Set from Favorites...** to specify a data value to be set.

Select this option to display the Favorites Chooser/Editor dialog box where you can:

- save an existing data value as an entry in your favorites list so that it can be re-used later in this debugging session
- take a data value already saved and use it to set the starting value for the specified watch.
Chapter 7
Reading and Writing Memory, Registers, and Flash

RealView Debugger includes options that enable you to work with registers and memory interactively during your debugging session. This chapter describes these options. It contains the following sections:

- About interactive operations on page 7-2
- Using the Memory/Register Operations menu on page 7-3
- Accessing interactive operations in other ways on page 7-5
- Working with Flash on page 7-6
- Examples of interactive operations on page 7-12.
7.1 About interactive operations

Use interactive operations to:
- set memory and registers
- patch assembly code (where supported by your debug target)
- read a file to memory
- write memory to a file
- verify memory against a file
- fill memory with a pattern of your choosing
- control Flash memory.

You can access all these features directly from the Debug menu. Selected operations are also available when you are working in panes. These are described in:
- Using the Memory/Register Operations menu on page 7-3
- Accessing interactive operations in other ways on page 7-5
- Working with Flash on page 7-6.

The last section in this chapter contains examples using interactive operations see:
- Examples of interactive operations on page 7-12.
7.2 Using the Memory/Register Operations menu

Use the Debug menu to carry out read and write operations on memory and registers:

1. Connect to your target and load an image, for example dhrystone.axf.
2. Select Debug from the Code window main menu to display the Debug menu.
3. Select Memory/Register Operations to display the menu shown in Figure 7-1.

This menu contains:

Set Memory...
Displays the Interactive Memory Setting dialog box where you can walk through memory and make changes where required.

Patch Assembly...
Enables you to patch assembly code during your debugging session. You can enter instructions in assembler format for patching directly into memory. You can use labels, including making new ones, and symbols. This is only available where supported by the underlying debug target, for example a DSP. This option is disabled by default.

Set Register...
Displays the Interactive Register Setting dialog box where you can walk through registers and make changes where required.

Upload/Download Memory file...
Displays the Upload/Download file from/to Memory dialog box where you can locate a specific file, of a given type, and read the contents into an area of memory, or write a memory range into the file for re-use, or verify that a memory range matches the file contents.

Fill Memory with Pattern...
Displays the Fill Memory with Pattern dialog box where you can specify a pattern that is used to write to a given area of memory.
Flash Memory Control...

Displays the Flash Memory Control dialog box where you can erase and write Flash memory. The Flash memory must be opened before trying to use this dialog box.
### 7.3 Accessing interactive operations in other ways

There are other ways to access interactive operations depending on where you are working:

- **From the Memory pane**
- **From the Stack pane**
- **From the Dsm tab.**

#### 7.3.1 From the Memory pane

If you are working in the Memory pane, you can access memory operations:

1. Select **View → Pane Views → Memory** to display the Memory pane.
2. Right-click on a memory cell, or byte, that is black or green, that is where the type is ROM, Flash, or modifiable, to display the context menu.
3. Select the required option, for example **Set Memory Interactive...** to display the Interactive Memory Setting dialog box.

#### 7.3.2 From the Stack pane

If you are working in the Stack pane, you can access memory operations:

1. Select **View → Pane Views → Stack** to display the Stack pane.
2. Right-click on a memory cell, or byte, to display the context menu.
3. Select the required option, for example **Set Memory Interactive...** to display the Interactive Memory Setting dialog box.

#### 7.3.3 From the Dsm tab

If the **Dsm** tab is visible, you can access memory operations:

1. Click on the **Dsm** tab to see the disassembly-level view.
2. Right-click on an address, to display the context menu.
3. Select the option **Patch Asm Interactive...** to patch assembly code.

   This is only available if supported by the underlying debug target. This option is disabled by default.
7.4 Working with Flash

To use RealView Debugger to control Flash memory on your chosen debug target, you must:

- configure your debug target to describe the Flash memory chip
- have access to an appropriate Flash Method (FME) file.

Depending on your current target, this might mean that you must first define the memory map to specify the Flash memory.

This section describes how to work with Flash memory during your debugging session. It includes:

- Flash definition files
- Flash Method files on page 7-7
- Flash examples on page 7-7
- Flash programming on page 7-7
- Using the Flash Memory Control on page 7-10.

7.4.1 Flash definition files

Files to enable you to use supported Flash devices are included in the root installation and are located in `\flash`:

**Board-specific files**

Assembler files start with `b_***`, for example `b_IntegratorAP.s`. Files starting with `board_***`, for example `board_intel_arm.ame`, contain the ASCII format information for an FME file.

These files include Flash memory programming files.

**Flash-specific files**

These programming files start with `f_***`, for example `f_intel_arm.s`, and `flash_***`, for example `flash_intel.ame`.

These files contain the algorithm for defining the Flash device and are used to create the FME file for your project.

To see how these files are used:

1. Start up RealView Debugger without connecting to a target.
2. Select **Project → Open Project...** to open the example project `\flash\examples\IntegratorAP\IntegratorAP.prj`.
3. Select **Project → Project Properties...** to display the Project Properties window.
4. Left-click on *ASSEMBLE=default in the List of Entries. This group is expanded and the contents are displayed in the Settings Values pane.

5. Right-click on *Sources and select Explore from the context menu. This shows the programming file used to create the FME file for the project.

6. Left-click on *BUILD in the List of Entries. This group is expanded and the contents are displayed in the Settings Values pane.

7. Right-click on *Pre_Post_Link and select Explore from the context menu. This shows the link commands used to include the Flash definition files for the project.

8. Select File → Close Window to close the Project Properties window without making any changes.

9. Select Tools → Build to create the FME file as defined by the project, that is flash_IntegratorAP.fme.

### 7.4.2 Flash Method files

FME files include code to:

- enable you to write to the Flash on your debug target
- perform read, write and erase operations
- describe the way that the Flash is configured on the bus.

Example files are included for supported Flash devices as part of the root installation.

### 7.4.3 Flash examples

The root installation contains a directory of examples for supported Flash devices. This area contains RealView Debugger project files that create FME files from assembler sources. All the Flash examples are located in `\flash\examples`.

### 7.4.4 Flash programming

Before you can use RealView Debugger to control a Flash device on your target, you must:

- describe the Flash memory chip in the memory map
- ensure that you have a correctly configured FME file.

The following example describes how to use the ARM Integrator FME file to program Flash memory on the Integrator/AP board. If you have another target board with a standard AMD, ATMEL, or Intel Flash device you must create a board-specific
assembler file and link that file to create an FME file before you can program the Flash memory. If you are using another type of Flash memory, you must also create the Flash programming routines.

The board specific assembler and Flash memory programming files are installed as part of the root installation, in `\flash\examples`, see Flash definition files on page 7-6 for details.

This example describes how to use the predefined Integrator/AP Flash configuration to write an image to the Flash memory on the Integrator/AP board, connected using Multi-ICE. The example is split into sections, which must be executed in this sequence:

1. **Defining your target**
2. **Programming the image into Flash** on page 7-9.

--- **Note**

If you program the Flash on an Integrator using this release of RealView Debugger, you bypass the **ARM Firmware Suite** (AFS) Flash library system information blocks. These blocks are used by the AFS Flash Library, and are stored at the end of each image written to Flash. If you rely on these blocks to keep track of what is in the Flash memory of your target, keep a record of the state and recreate it after working through the example.

---

**Defining your target**

To configure the Flash target:

1. Start up RealView Debugger without connecting to a target.
2. Select `File → Connection → Connect to Target...` to display the Connection Control window.
3. Right-click on the entry `Multi-ICE` and select **Connection Properties...** from the context menu.
   This displays the Connection Properties window where you can view configuration settings stored in your board file.
4. Click on the entry `CONNECTION=Multi-ICE`, in the left pane, to display the settings values in the right pane.
5. Right-click on the entry `BoardChip_name` and select **AP** from the context menu. This means that the Integrator/AP board file is used for this connection.
6. Select `File → Save and Close` to close the Connection Properties window.
7. Connect to the target using the Connection Control window.

8. Click on the Log tab in the Output pane to see that RealView Debugger is using the Integrator/AP board file.

9. Select View → Pane Views → Memory Map to display the Map tab where you can see the Flash memory on the Integrator/AP board.

In this example, memory has been mapped, using a board/chip definition file, to declare Flash. See the chapter describing configuring custom targets in RealView Debugger v1.6 Target Configuration Guide for details of configuring your target this way.

**Programming the image into Flash**

To program the image, you ask RealView Debugger to write to the Flash memory region that you have defined in the board file. The Integrator Flash starts at memory address 0x24000000, so to write an image to Flash:

1. Build an image compiled to run with code at 0x24000000 and that has data in RAM.

   This example uses the dhrystone program stored in the \Examples directory in your root installation. Open the project and rebuild using modified linker options.

   Set the Link_Advanced settings in the BUILD group:
   • \Ro_base = 0x24000000
   • \Rw_base = 0x8000.

2. Select File → Load Image... to load your image. This displays the Load File to Target dialog box where you can locate the required image.

3. Click Open in the Load File to Target dialog box. This displays the Flash Memory Control dialog box, shown in Figure 7-2 on page 7-10.
4. Click **Write** to program the image into Flash.

5. Click **Close** to close the Flash Memory Control dialog box.

6. Click on the **Cmd** tab in the Output pane to see the Flash operations.

7. Select **View → Pane Views → Memory Map** to display the **Map** tab where you can see the Flash memory on the Integrator/AP board.

### 7.4.5 Using the Flash Memory Control

The Flash Memory Control dialog box consists of a display list, a read-only data field, a Flash Log, and control buttons:

**Flash:** Click this button to get details about the Flash. The data field next to the **Flash** button describes the type of Flash being used.

**Open Flash Blocks:**

- **AllOn** Selects all entries in the Open Flash Blocks list as indicated by a check in the accompanying check box. This enables you to carry out operations on all the open blocks.
- **AllOff** Unselects all entries in the Open Flash Blocks list as indicated by no check in the accompanying check box.
Write  Writes data to the specified blocks of Flash.

Erase  Erases every specified block of Flash. This normally sets every byte to 0x00 or 0xFF depending on the type of Flash being used.

Cancel  Abandons any changes made to the specified blocks of Flash.

Cancel All  Abandons all changes to the Flash contents.

Details  Displays an information box describing the type of Flash being used.

Erase Block before Write
Select this check box to erase the Flash block before performing the write operation.

Verify Block after Write
Select this check box to verify the Flash block, against the data source, after performing the write operation.

Use Current values for Unspecified data in block
Specifies that the original contents should be maintained unless modified by the current operation. If unselected, the erase values are used.
This option is unselected by default.

Flash Log:
Displays a log of operations carried out on the selected Flash blocks.

Close  Closes the Flash Memory Control dialog box.

Help  Displays online help for this dialog box.

See Setting Flash memory on page 7-20 for an example of interactive Flash memory operations.
7.5 Examples of interactive operations

This section contains examples showing how to perform interactive operations on memory and registers:

- Setting memory
- Setting registers on page 7-14
- Downloading memory to a file on page 7-15
- Comparing memory with file contents on page 7-17
- Filling memory with a pattern on page 7-18
- Setting Flash memory on page 7-20.

7.5.1 Setting memory

To set memory contents:

1. Connect to your target and load an image, for example dhrystone.axf.
2. Click on the Src tab to view the source file dhry_1.c.
3. Set a simple breakpoint by double-clicking on line 150, Proc_4();.
4. Click Go to start execution.
5. Enter 5000 when asked for the number of runs.
   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.
6. Select View → Pane Views → Memory to display the Memory pane. Start addresses can be set using in-place editing or using the context menu.
7. Right-click in the first address in the window to display the context menu.
8. Select Set New Start Address... and enter 0x000088A0 as the new start address.
9. Highlight the first byte at this address, that is 0xBF.
10. Right-click and select Set Memory Interactive... from the context menu to display the Interactive Memory Setting dialog box, shown in Figure 7-3 on page 7-13.
11. Enter the required memory settings:

**Type:** Select the display format. See *Formatting options* on page 6-13 for details of the memory formats.

**Addr:** The address where the memory setting starts. Depending on the method used to display this dialog box, this field is already populated, as in this example. The address must be entered in hexadecimal format, for example 0x000088A0.

**Value:** This read-only data field shows the current value, in hexadecimal and decimal formats, at the specified memory location.

**Enter New Value:**
Enter the value to be set at the current location, for example 0x08 or 8 (decimal).

If the Memory pane is configured to update automatically, clicking Set immediately updates the memory contents. This is the default setting in the Pane menu.

If you press Enter with no value in the Value data field, RealView Debugger moves automatically to the next, or previous, location.

**Next Addr**
Moves the target address to the next location by adding 1 to the address displayed in the Addr data field. This depends on the size of the current type.
Prev Addr
Moves the target address to a new location by subtracting 1 from the address displayed in the Addr data field. This depends on the size of the current type.

Clear New
Automatically clears any value entered in the Enter New Value data field ready to accept another value. By default, this feature is enabled.

Auto Inc Addr
If selected, this radio button instructs RealView Debugger to increment the target address automatically ready to accept a new setting.

Auto Dec Addr
If selected, this radio button instructs RealView Debugger to decrement the target address automatically ready to accept a new setting.

Log: Displays a log of the changes you have made. This log is shown when you next display the dialog box.

12. Click Close to close the Interactive Memory Setting dialog box.

Changed values are displayed in the Memory pane in the usual way. That is, updated values are displayed in dark blue or light blue, depending on when they last changed.

7.5.2 Setting registers

To set register contents:

1. Select File → Reload Image to Target to reload the image dhrystone.axf.
2. Click on the Src tab to view the source file dhry_1.c.
3. Set a simple breakpoint by double-clicking on line 150.
4. Click Go to start execution.
5. Enter 5000 when asked for the number of runs.
   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.
6. Select View → Pane Views → Registers to display the Register pane.
7. Select Debug → Memory/Register Operations → Set Register... from the Code window main menu to display the Interactive Register Setting dialog box. This dialog contains almost the same controls as the Interactive Memory Setting dialog box described in Setting memory on page 7-12.
8. Set up the required register settings:

**Register:** Enter the register to change, for example @R4. Press Enter to confirm your choice.
   If required, use the drop-down arrow to select a previously used register from the stored list.

**Value:** This read-only data field shows the current value, in hexadecimal and decimal formats, for the specified register.

**Enter New Value:**
   Enter the value to be set, in hex or decimal, for example 0xCC4.
   All changed registers are displayed in blue.

9. Click **Close** to close the Interactive Register Setting dialog box.

### 7.5.3 Downloading memory to a file

To download a memory range into a file:

1. Select **File → Reload Image to Target** to reload the image dhrystone.axf.
2. Click on the **Src** tab to view the source file dhry_1.c.
3. Set a simple breakpoint by double-clicking on line 150.
4. Click **Go** to start execution.
5. Enter 5000 when asked for the number of runs.
   The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.
6. Select **View → Pane Views → Memory** to display the Memory pane.
7. Select **Debug → Memory/Register Operations → Upload/Download Memory file...** from the Code window main menu to display the Upload/Download file from/to Memory dialog box, shown in Figure 7-4 on page 7-16.
8. Set up the required memory settings:

**Load File into Memory**
Select to load a file to memory. This enables RealView Debugger to access the specified file, read the contents, and write them to the given memory location.

**Save Memory into File**
Select to save memory to a file. This enables RealView Debugger to access the specified memory block, read the contents, and write them to the given file.

**Verify Memory and File**
Select to compare the contents of a memory block with a specified file. The results of the comparison are reported in the **Cmd** tab in the Output pane.

**File:** Enter here the full pathname of the file to use to read/write memory values.

**Type of File:**
Enter the data type to be used in the specified file where:

- **OBJ** specifies an object file in the standard executable target format, for example ARM-ELF for ARM-based targets
- **raw** specifies a data file as a stream of 8-bit values
- **ascii** specifies a space-separated file of hexadecimal values.

**Location:** Define the start location of the memory block.
When writing memory, specify a range as an address range or as a start address and length, for example:
Reading and Writing Memory, Registers, and Flash

0x88A0 .. 0x8980
0x88A0 .. +0x14
If required, use the drop-down arrow to select a previously used location from the stored list.

--- Note ---
If you are reading from a file to memory, you must specify a start location. The range can be left blank where the data type is not binary.
If you are writing to a file from memory, you must specify a start location and a range.

---

Apply
Click this button to create and write the specified file, if this is a new file, or to open and read the contents of a file to the specified memory location(s).

Close
Click this button to close the Upload/Download file to/from Memory dialog box.

--- Note ---
If you are writing memory to a file and the specified file already exists, RealView Debugger warns of this and asks for confirmation before overwriting the file contents.

---

RealView Debugger warns you if the memory transfer is going to take a long time to complete. When reading or writing memory contents, you must be aware that:

- There is no limit on the size of file that RealView Debugger can handle.
- The time taken to complete the operation depends on the access speeds of your debug target interface.

7.5.4 Comparing memory with file contents

To verify a file:

1. Select Debug → Memory/Register Operations → Upload/Download Memory file... from the Code window main menu to display the Upload/Download file from/to Memory dialog box (see Figure 7-4 on page 7-16).

2. Select Verify File and Memory.

3. Specify the file to be compared.

4. Specify the memory range to be compared.

5. Click Apply to compare the file contents with the specified memory block.
6. Click on the **Cmd** tab in the Output pane and view the results, for example:

```
verifyfile,ascii,gui "C:\RealView Debugger\Test_files\memory_file_3"
```

Mismatch at Address 0x000088B6: 0x8E vs 0x8F

The first mismatch is identified and the location reported. Any mismatches after this location are not reported.

7. Click **Close** to close the Upload/Download file from/to Memory dialog box.

### 7.5.5 Filling memory with a pattern

To fill a specified area of memory a predefined pattern:

1. Select **File → Reload Image to Target** to reload the image dhrystone.axf.
2. Click on the **Src** tab to view the source file dhry_1.c.
3. Set a simple breakpoint by double-clicking on line 150.
4. Click **Go** to start execution.
5. Enter **5000** when asked for the number of runs.

The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the PC when execution stops.

6. Select **View → Pane Views → Memory** to display the Memory pane.

7. Select **Debug → Memory/Register Operations → Fill Memory with Pattern...** from the Code window main menu to display the Fill Memory with Pattern dialog box, shown in Figure 7-5.

![Fill Memory with Pattern dialog box](image)

#### Figure 7-5 Fill Memory with Pattern dialog box

8. Set up the required memory settings:

   **Start:** Enter the start address for the memory range to be filled, for example 0x88A0.

   If required, use the drop-down arrow to select a previously used start address from the stored list.
End/Len: By default, the memory area that is filled is defined by a start address and a length. Enter the length of the memory block to be filled in this data field, for example 14 (decimal).
The specified length must be given relative to the data type given in the Size data field specified below.
If the Use Length (Count) check box is unselected, you can specify the address that marks the end of the memory block.

Size: Enter the data type to be used in the specified file where:
- natural indicates the format specified as native for the debug target
- byte indicates support for 8-bit signed and unsigned byte form
- half-word indicates support for 16-bit signed and unsigned halfwords
- long-word indicates support for 32-bit signed and unsigned words.

Use Length (Count)
By default, the memory block to be filled is defined by a start address and the length. If this check box is unselected you can specify an address to mark the end of the filled block.

Pattern: Enter the pattern to be used as the fill, for example:
- “AB”
- 0,1,0,1,0

OK Click OK to confirm your settings and to close the dialog box.
Memory contents are rewritten and the Memory pane is automatically updated with changed values displayed in blue.

Cancel Click this button to close the Fill Memory with Pattern dialog box without changing memory contents.

When filling memory blocks, you must be aware that:
- All expressions in an expression string are padded or truncated to the size specified by the Size value if they do not fit the specified size evenly.
- If the number of values in an expression string is less than the number of bytes in the specified address range, RealView Debugger repeats the pattern and so might fill an area in excess of the specified block, for example specify a pattern of 10 bytes and a fill area of 16 bytes. RealView Debugger repeats the pattern twice and so fills a block of 20 bytes.
- If more values are given than can be contained in the specified address range, excess values are ignored.
If a pattern is not specified, RealView Debugger displays an error message.

7.5.6 Setting Flash memory

Flash memory blocks are opened for access when you write to Flash. This displays the Flash Memory Control dialog box, for example when you load an image.

To write to Flash memory interactively:

1. Connect to your target, load an image, and write to Flash, as described in Working with Flash on page 7-6.

2. Select View → Pane Views → Memory to display the Memory pane.
   Start addresses can be set using in-place editing or using the context menu.

3. Right-click in the first address in the window to display the context menu.

4. Select Set New Start Address... and enter 0x24000000 as the new start address.
   This is colored green, indicating Flash.

5. Right-click in the first byte at this address.

6. Select Set Value... from the context menu.

7. Enter the new value 0xA0 at the prompt.

8. Click Set to confirm.
   This displays the Flash Memory Control dialog box to enable you to access the open Flash, shown in Figure 7-2 on page 7-10.

9. Click Write to write to the chosen Flash location. Monitor the changes in the Memory pane as memory is updated. The Flash Log confirms the Flash operation.

10. Click Erase to erase the chosen Flash block.

11. Click Flash to display details of the Flash memory, shown in Figure 7-6 on page 7-21.
12. Click **Erase** to erase the chosen Flash block.

13. Click **Close** to close the Flash Memory Control dialog box.

--- Note ---

You can also select **Debug → Memory/Register Operations → Flash Memory Control...**, from the Code window main menu, to display the Flash Memory Control dialog box during your debugging session.
Chapter 8
Working with Browsers

RealView Debugger provides list browsers to help with debugging tasks and monitor your program during execution. This chapter describes how to access these lists from the Code window. It contains the sections:

- Using browsers on page 8-2
- Browsing modules and files on page 8-3
- Browsing functions on page 8-6
- Browsing variables on page 8-10
- Specifying browser lists on page 8-13
- Browsing C++ classes on page 8-15
- Other routes to the browsers on page 8-17.
8.1 Using browsers

Browsers enable you to search through your source files to look for specific structures and to monitor their status during program execution. Browsers are available for:

- project modules and files
- functions
- variables
- C++ classes.

RealView Debugger uses scope to determine the value of a symbol. Any symbol value available to a C or C++ program at the current PC is also available to RealView Debugger.

Variables can have values that are relevant within:

- a specific class only, that is class scope
- a specific function only, that is local scope
- a specific file only, that is static global scope
- the entire process, that is global scope.

For full details on scope and scoping rules see the chapter describing working with the CLI in RealView Debugger v1.6 Command Line Reference Guide.

8.1.1 Accessing list browsers

To access the browsers, select the Find menu from the Code window main menu.

This menu displays the main searching facilities available from RealView Debugger. It also includes:

- Module/File List
- Function List
- Variable List.

Where supported by your debug target, you can also access a list of memory mapped registers when you use the Set Address/Data Break/Tracepoint dialog box to set a breakpoint.

In addition, RealView Debugger includes a Symbol browser to view C++ class objects that can be accessed through the Symbol Browser pane. Either:

- Select View → Pane Views → Symbol Browser from the default Code window main menu.
- Click on the Pane Content menu in a chosen pane, for example the Watch pane, and select Symbol Browser.
8.2 Browsing modules and files

Using the Module/File browser enables you to examine the different files and modules that go to make up your program and how these components are accessed during program execution. In this way you can locate errors during your debugging session.

To display the Module/File List, shown in Figure 8-1, select Find → Module/File List... from the Code window main menu.

The Module/File List dialog box displays, in order of appearance, all the modules and files in the current program. Each entry in the list shows the module name and then the filename, if known, for example:

@dhrystone\DHRY_2 - dhry_2.c

The program name is attached at the start using @, for example @dhrystone\.

Module names qualify symbolic references. The module name is usually the filename without the extension. All module names are converted to uppercase by RealView Debugger. If the extension is not standard, the extension is preserved, and the dot is replaced with an underscore, for example sample_arm.c is converted to SAMPLE_ARM, and sample_arm.h is converted to SAMPLE_ARM_H.

If two modules have the same name then RealView Debugger appends an underscore followed by a number to the second module, for example SAMPLE_1. If there is a third module this becomes SAMPLE_2 and so on for any additional modules.

Following this convention avoids any confusion with the C dot operator indicating a structure reference.

This section describes:

- Specifying the list on page 8-4
- Scoping to a module or file on page 8-4
- Closing the browser on page 8-5.
8.2.1 Specifying the list

When you first open the Module/File List dialog box, the list entries are determined by the default search entry ISearch but you can decide which modules and files are displayed by applying a search filter.

See Specifying browser lists on page 8-13 for details of how to specify the list for the chosen browser.

8.2.2 Scoping to a module or file

To scope to a module:

1. Click on the Src tab to view the source file dhry_1.c.
2. Select Find → Module/File List... to display the Module/File List dialog box shown in Figure 8-1 on page 8-3.
3. Select a module from the list.
4. Click Scope to scope to that module.

The Src tab is updated to show that the scope is forced, as illustrated in Figure 8-2.

The location in the source file is identified by:

- placing a blue pointer to the left of the line number in the File Editor pane
- enclosing the line of code in a red box showing the location of the PC.

If the file is not already open in the File Editor pane, RealView Debugger opens it automatically, the window scrolls to the right place, and the scope is adjusted. The Module/File List dialog box closes.
8.2.3 Closing the browser

If you scope to a new module or file, the Module/File List dialog box closes automatically. Otherwise, click **Cancel** to exit the browser without adjusting the scope.
8.3 Browsing functions

Use the Function browser to examine the different functions that go to make up your program.

To display the Function List, shown in Figure 8-3, select Find → Function List... from the Code window main menu.

![Function List dialog box](image)

**Figure 8-3 Function List dialog box**

The Function List dialog box lists all the functions, ordered by module name, in the current program. Each entry in the list shows the filename, if known, and then the function name, for example:

```
DHRY_2\Func_3 of @dhrystone
```

This section describes:

- Specifying the list
- Refining the list on page 8-7
- View details of a function on page 8-7
- Scoping to a function on page 8-8
- Setting a breakpoint on page 8-8
- Closing the browser on page 8-9.

8.3.1 Specifying the list

When you first open the Function List dialog box, the list entries are determined by the default search entry ISearch but you can decide which modules and files are displayed by applying a search filter.

See Specifying browser lists on page 8-13 for details of how to specify the list for the chosen browser.
8.3.2 Refining the list

The Function List dialog box contains check boxes that enable you to refine what is displayed in the list box:

- **Publics**: Displays global or public functions with scope over all parts of the program.
- **Statics**: Displays static functions.
- **Labels**: Displays code labels with scope over the entire function.

8.3.3 Viewing details of a function

Highlight a function in the display list. The Function List dialog box contains controls to display more details of this function or to perform specific debugging activities:

- **Disasm**: Displays the memory address in hexadecimal and assembly code in the `Dsm` tab starting at the specified memory location.
- **Source**: Displays the source code in the corresponding `Src` tab beginning at the specified line number or procedure name.
- **Break**: Sets a software breakpoint at the specified function, defined as a location in the image.
- **GoTo**: Sets a temporary breakpoint at the specified function. The program then executes from the current position of the PC. When execution reaches the breakpoint it stops.
  
  The temporary breakpoint exists only for the duration of this run and so is not shown in the Break/Tracepoints pane. Similarly, there is no red breakpoint marker shown in the source file. If the program stops before it reaches the temporary breakpoint, you must reinstate it before restarting the run.

- **Type**: Displays type information for the selected function. This information is displayed in a style similar to the source language.

- **Info**: Submits a `C` expression `CEXPRESSION` command to calculate the value of a given expression by calling the specified target function.
  
  The function is converted into a debugger call macro, and strings and arrays passed to the target function are copied onto a stack maintained by RealView Debugger. A function called in this way behaves as though it had been called from your program.
Note

Target calls are not supported by all debug environments.

Results are displayed in either floating-point format, address format, or in decimal, hexadecimal, or ASCII format depending on the type of variables used in the expression.

SetPC

Submits a SETREG command to change the contents of a specified register identified by @ followed by the register name.

The Code window scrolls to the specified function and the red box shows the location of the PC.

8.3.4 Scoping to a function

With a list of functions displayed in the Function List you can select an entry and click the Scope+Close button to scope to that function and so adjust the context. This displays a confirmation message in the Output pane.

If the file is not already open in the File Editor pane RealView Debugger opens it automatically, the window scrolls to the right place and the scope is adjusted. The Function List dialog box closes.

8.3.5 Setting a breakpoint

You can use the Function List dialog box to set a breakpoint on a chosen function:

1. Highlight a function in the display list.
2. Click Break to set a breakpoint.

A cut-down version of the Function browser is also available to set a breakpoint from the Debug menu:

1. Click on the location of your breakpoint in your code view.
2. Select Debug → Simple Breakpoints → Set from Function/Label list... to display the Function Breakpoint/Profile Selector dialog box.

Because the browser is used only to make a selection, there are no controls for debugging operations.

The Function Breakpoint/Profile Selector does not provide a record of breakpoints already set, that is, when you next open this dialog box existing breakpoints are not checked.
8.3.6 Closing the browser

If you scope to a new module or file, the Function List dialog box closes automatically. Otherwise, click **Cancel** to exit the browser without adjusting the scope.
8.4 Browsing variables

Use the Variables browser to examine the different variables used in your program.

To display the Variable List dialog box, shown in Figure 8-4, select **Find → Variable List...** from the Code window main menu.

![Variable List dialog box](image)

**Figure 8-4 The Variable List**

The Variable List dialog box shows all the variables in the current program. By default, the **Sort** check box is unselected and so the variables are given in order of occurrence.

Each entry in the list shows the variable name followed by the program name attached using @, for example:

Ch_1_Glob of @dhrystone

Including local variables adds the filename, if known, to the list:

DHRY_2\Proc_8\Int_Index local of @dhrystone

This section describes:

- **Specifying the list**
- **Refining the list** on page 8-11
- **Viewing details of a variable** on page 8-11
- **Closing the browser** on page 8-12.

8.4.1 Specifying the list

When you first open the Variable List dialog box, the list entries are determined by the default search entry **ISearch** but you can decide which modules and files are displayed by applying a search filter.
8.4.2 Refining the list

The Variable List dialog box contains check boxes that enable you to refine what is displayed in the list box:

<table>
<thead>
<tr>
<th>Publics</th>
<th>Displays global or public variables with scope over all parts of the program.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statics</td>
<td>Displays static variables.</td>
</tr>
<tr>
<td>Labels</td>
<td>Displays code labels with scope over the entire function.</td>
</tr>
<tr>
<td>Locals</td>
<td>Displays local variables with scope within the current function.</td>
</tr>
</tbody>
</table>

8.4.3 Viewing details of a variable

Highlight a variable in the display list. The Variable List dialog box contains seven buttons to display more details of this variable in the Output pane or to carry out specific debugging activities:

<table>
<thead>
<tr>
<th>Print</th>
<th>Displays the value, in decimal, of the variable in the current procedure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print Hex</td>
<td>Displays the variable value in hexadecimal.</td>
</tr>
<tr>
<td>Watch</td>
<td>Click to copy the selected variable into the Watch pane so that you can view the watched data and see how the value changes during program execution.</td>
</tr>
<tr>
<td>PrintType</td>
<td>Displays type information for the selected variable. This information is displayed in a style similar to the source language.</td>
</tr>
<tr>
<td>Info</td>
<td>Displays type information for the selected variable including name, data type, storage class, and memory location.</td>
</tr>
<tr>
<td>Print+Close</td>
<td>Displays the value, in decimal, of the variable in the current procedure. When the PRINTVALUE command has completed, the Variable List dialog box closes and control returns to the Code window.</td>
</tr>
</tbody>
</table>
Break+Close

The behavior of this button depends on your debugging environment. Possible results are:

- Sets a breakpoint at the specified function, defined as a location in the image. The position of the breakpoint is indicated by a red disc. If this software execution break is reached while the program is running, RealView Debugger halts execution of the image. When the breakpoint has been inserted, the Variable List dialog box closes and control returns to the Code window.

- Displays the Set Address/Data Break/Tracepoint dialog box with the location filled in.

8.4.4 Closing the browser

Click Close to exit the browser.
8.5 Specifying browser lists

When you first open a browser, the list entries shown in the dialog box are determined by the default search entry \texttt{ISearch} but you can decide what contents are displayed by applying a search filter. To change the search mechanism either:

- click on the drop-down arrow to display the search list and select a search
- highlight the contents, for example \texttt{ISearch}, and press F or I to toggle the contents.

This section describes:

- Specifying a list
- Applying a filter.

8.5.1 Specifying a list

To specify the search:

\texttt{ISearch} \quad With this selected, the list entries are created using the default search mechanism for variable names. Enter a partial name to move the highlight to the first matching occurrence. The search is case insensitive.

\texttt{Filter} \quad Use a filter to limit the search to find only those symbols that match certain criteria, see Applying a filter for details.

\textbf{Text entry field}

Enter here the filter to be used in the search and then press Enter. The filter enables you to search using UNIX rules as with \texttt{ls}. You can enter a list of search rules by combining different operators, see Applying a filter for details of how to apply a search filter.

\textbf{Sort} \quad Select this check box to order the display list in alphabetical order by name. The sort is case sensitive, that is uppercase and lowercase are treated as identical. This check box is unselected by default.

If you change the default search entry to \texttt{Filter} and then enter a filter to set the display criteria, these settings are maintained when you close, or cancel, the browser.

8.5.2 Applying a filter

Using a filter enables you to narrow down the search when displaying the list of modules and files. Table 8-1 on page 8-14 shows the metacharacters you can use to specify the filter rule or rules. When entering a filter, characters are case sensitive, for example the filter \texttt{*DHRY*} returns a list of five modules but \texttt{*dhry*} returns an empty list.
When you have completed the filter, press Enter and the list is refreshed. By repeatedly entering filters, and pressing Enter, you can refine the search to focus on selected modules or files.

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>This operator matches any character or number of characters, for example <em>DHRY</em> matches MY_DHRYSTONE_H but not Dhrystone_H or MY_DHR.</td>
</tr>
<tr>
<td>?</td>
<td>This operator matches any single character, for example *DHRY_?. matches MY_DHRY_A but not MY_DHRY_AB or DHRY_B.</td>
</tr>
<tr>
<td>[...]</td>
<td>List operators enable you to define a set of items to use as a filter. The list items must be enclosed by square brackets, for example <em>[HN]</em> matches DHRY_H and UNNAMED_1 but not STDLIB. An empty list ([ ]) returns no results.</td>
</tr>
<tr>
<td>^</td>
<td>This operator is used inside a list, to represent a NOT action, for example <em>.^[2]</em> matches DHRY_1 but not DHRY_2.</td>
</tr>
<tr>
<td>-</td>
<td>Range operators enable you to define a range of items to use as a match. The range must be enclosed within square brackets, for example <em>_.[A-Z]</em> matches DHRY_H but not DHRY_1 whereas <em>.^[A-Z]</em> matches UNNAMED_1 but not DHRY_H.</td>
</tr>
<tr>
<td></td>
<td>Used as the first character in a filter, this operator means do not match. For example, <em>.SAM</em> -<em>HOST</em> means match all names containing the string SAM except those that contain the string HOST.</td>
</tr>
</tbody>
</table>
### 8.6 Browsing C++ classes

Use the Symbol Browser pane to examine C++ classes in your application program.

To examine C++ classes either:

- Select the menu option View → Pane Views → Symbol Browser from the Code window main menu.

- Move the focus to the chosen pane, for example the Watch pane, click the Pane Content menu, and select Symbol Browser from the available options.

An example display is shown in Figure 8-5.

![Figure 8-5 C++ symbols in the Symbol Browser pane](image)

This section describes:

- Viewing details of a class
- Viewing details of a function on page 8-16.

#### 8.6.1 Viewing details of a class

Colored icons are used to identify different components within a class:

- **Filled stack + arrow**
  
  This magenta icon indicates a function which is a declared member of the parent class.

- **Filled stack**
  
  The magenta filled stack indicates that a member function of the parent class is both declared and defined. These members are real in that they are called during execution.
Hollow stack

Where magenta stacks are hollow, they indicate that a member function of the parent class is declared but not defined. These members are virtual in that they are not called during execution.

Filled block

The filled blue block indicates a data object that is manipulated by a class function using operators, or methods, defined in the class.

Right-click on a chosen class to display the Class menu. This contains:

Find Class Definition...

Displays the Find in Files dialog box where you can specify the class details and so locate the definition in your source code.

Use this dialog box to locate class definitions when these are not provided by the compiler.

See Chapter 13 Searching and Replacing Text for details on using the Find in Files dialog box.

Properties

Displays a text description of the item under the cursor.

8.6.2 Viewing details of a function

Right-click on a function to display the Function menu. This contains:

Show Function Definition

Scopes to the selected function. This option is enabled for defined functions identified by a magenta filled stack icon.

Set Break

Sets a breakpoint at the selected function. This option is enabled for defined functions identified by a magenta filled stack icon.

Properties

Displays a text description of the item under the cursor.
8.7 Other routes to the browsers

You can access cut-down browsers from other routes when working with RealView Debugger. These alternative access paths enable you to use the browsers to select a program structure for:

- running a program trace to display the procedure calling chain from the main program to the current procedure
- setting watches to monitor changes in contents of specific locations.

To view a browser this way:

1. Select View → Pane Views → Watch to display the Watch pane.
2. Click on the Pane menu.
3. Select Enter New Expression... to display the expression prompt box.
4. Click on the drop-down arrow to display the options list.
5. Select the required browser, for example <Variable list...>.
   Because the browser is used only to make a selection, there are no controls for debugging operations such as PrintType, Break, or Info.
6. Highlight your choice in the browser display list, and click Select.
   This closes the browser and enters the chosen expression into the data field
7. Click Set to confirm this choice and close the prompt box.

Other dialog boxes include a drop-down arrow that enables you set the required expression using different browsers, for example click the Pane menu from the Memory pane to choose the option Set New Start Address....
Chapter 9
Working with Macros

In RealView Debugger, a macro is a C-like function that is invoked by entering a single command using the macro name. This section describes how to define macros for use during your debugging session, how to save and edit your macros, and how to use predefined macros that form part of RealView Debugger. It contains the following sections:

- About macros on page 9-2
- Using macros on page 9-7
- Source patching with macros on page 9-13
- Macro language on page 9-18
- Predefined macros on page 9-26.
9.1 About macros

Macros are interpreted C code running on the workstation with access to target memory and symbols, user-defined debugger symbols, in host or target memory, and debugger functions. Macros can access debugger variables, external windows and programs, and can be attached to breakpoints, aliases, and windows.

A macro can contain:
- a sequence of expressions
- string formatting controls
- statements
- calls to other macros
- predefined macros
- target functions
- debugger commands.

You can define and use macros at any time during a debugging session to use the commands or statements contained in the macro. You call the macro with a single command using the name. The macro definition might contain parameters that you change each time the macro is called.

When a macro is defined, you can use it as:
- a complex command or in an expression
- an attachment to a breakpoint to create complex breakpoint condition testing
- an attachment to a window to display information in it.

This section gives an overview of macros in RealView Debugger. It includes the following sections:
- Properties of macros on page 9-3
- Debugger commands in macros on page 9-3
- Defining macros on page 9-4
- Calling macros on page 9-4
- Macro return values on page 9-5
- Using macros with breakpoints on page 9-5
- Attaching macros on page 9-6
- Stopping macros on page 9-6.

RealView Debugger also includes a series of predefined macros that you can use during your debugging session. See Predefined macros on page 9-26 for a full description.
9.1.1 Properties of macros

Macros can:

• have return values
• contain C expressions
• contain certain C statements
• have arguments
• define macro local variables
• use conditional statements
• call other macros and predefined macros
• be used in expressions, where they return values
• reference target variables and registers
• reference user-defined variables, in debugger or target memory
• execute most debugger commands
• be defined in a debugger include file.

Macros cannot:

• be recursive
• define global variables
• define static variables
• define other macros.

9.1.2 Debugger commands in macros

You can define a macro that contains a sequence of debugger commands. When used in this way, the command must be enclosed by dollar signs ($), for example:

```c
$printf 50, "the important variable is %d\n", var1$;
```

Macros containing commands are similar to command files and can be used for setting up complex initialization conditions. These macros are executed by entering the macro name and any parameters on the RealView Debugger command line.

The following commands cannot be used inside a macro:

• ADD
• DEFINE (unless it is the macro definition itself)
• DELETE
• HELP
• HOST (this command is only available on UNIX)
• INCLUDE
• QUIT.
Because macros can return a value, they can also be used in expressions. When the macro executes, the return value is used according to the return type.

**Attaching macros**

Macros can also be invoked as actions associated with:

- a window
- a breakpoint
- deferred commands, for example `BGLOBAL`.

In this case, execution-type commands cannot be used inside a macro:

- `GO`
- `G0STEP`
- `STEPLINE`, `STEPO`
- `STEPINSTR`, `STEPOINSTR`.

If you require a conditional breakpoint that performs an action and then continues program execution, you must use the breakpoint `continue` qualifier, or return 1 from the macro call, instead of the `GO` command.

### 9.1.3 Defining macros

You can define a macro outside RealView Debugger using an editor and load the macro definition file in with the `INCLUDE` command. The number of macros that can be defined is limited only by the available memory on your workstation.

You can also create, edit, save, and delete macros from the desktop using the Debug menu, as demonstrated in Using macros on page 9-7.

See *Macro language* on page 9-18 for a full description of the macro syntax.

### 9.1.4 Calling macros

Macros are called from the RealView Debugger command line. The call consists of the macro name followed by a set of parentheses containing the macro arguments, separated by commas. Macro names are case sensitive and must be entered as shown in the definition. The macro arguments are converted to the types specified in the macro definition. If RealView Debugger cannot convert the arguments it generates an error message.

Examples of macro calls are:

- `mytext(var)` Calls the macro named `mytext` with the argument `var`.
count(7)  Calls the macro named count with the parameter 7.

You can define a macro with a name that is identical to a command or keyword used by RealView Debugger. You can then use the macro name in an expression and submit it on the command line where it is interpreted correctly.

If, however, you submit the macro name as a command, RealView Debugger cannot identify it as a macro. To overcome this, use the prefix MACRO when entering macro names that might conflict with debugger keywords or command names:

MACRO macro_name()

Macros take higher precedence than target functions. If a target function and a macro have the same name, the macro is executed unless the target function is qualified. For example, strcpy is a predefined debugger macro, while PROG\strcpy is a function within the module PROG. The predefined macro is referenced as strcpy(dest,src), while PROG\strcpy(dest,src) refers to the function within PROG.

9.1.5  Macro return values

You can use macro return values to control what action RealView Debugger takes when a conditional breakpoint is triggered. If the macro returns a true value, that is nonzero, RealView Debugger continues program execution. If a macro returns a false value, that is zero, RealView Debugger stops program execution.

The type of the macro return value is specified by return_type when you define the macro. If return_type is not specified, then type int is assumed.

9.1.6  Using macros with breakpoints

When you set a breakpoint, you can also associate a macro with that breakpoint for complex break conditions. You can also attach predefined macros to breakpoints, for example by using the context menu option Set BreakIf... from the Src tab.

In this way you can test your program variables and decide whether execution stops or continues after the breakpoint has been triggered. When you have attached a macro to a breakpoint it can be executed every time the breakpoint is triggered.

You can use conditional statements in your macro to change the execution path when the breakpoint is triggered depending on variables on the debug target system or on the host workstation. This enables you to control program execution during your debugging session or when there is no user intervention. You can also use high-level expressions in macros. Combining these conditional statements and expressions enables you to patch your source program, see Source patching with macros on page 9-13.
Breakpoint macros can be used to fill out stubs, such as I/O handling, and also to simulate complex hardware.

For an example of attaching a macro to a breakpoint and using it to control program execution see *Attaching macros to breakpoints* on page 4-21.

**9.1.7 Attaching macros**

In addition to breakpoints, you can attach macros to:

- aliases, for example `ALIAS my_alias=my_macro()`
- windows, for example `VMACRO 250, my_macro()`.

**9.1.8 Stopping macros**

When macros are run as commands they are queued for execution just like any other debugger command when your program is executing. Click the **Command cancel** toolbar button to cancel the last command entered onto the queue. This can be used to stop any macro that is running. This does not take effect until the previous command has completed and so any effects might be delayed.

Click the **Stop Execution** button to stop a macro that is attached to a breakpoint.
9.2 Using macros

This section shows you how to start using macros by working through an example. It contains the following sections:

- Creating a macro
- Viewing a macro on page 9-8
- Testing a macro on page 9-9
- Editing a macro on page 9-9
- Copying a macro on page 9-11
- Deleting a macro on page 9-11
- Calling a macro on page 9-12.

9.2.1 Creating a macro

Complete the following steps to create a macro for use with a debug target:

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

2. Select **Debug** → **Add/Edit Debugger Macros**... to display the Add/Edit Macros dialog box.
   
   When you first open this dialog box, the Existing Macros display list is empty because no macros have been defined or loaded into RealView Debugger.
   
   The Macro Entry Area gives advice on how to use the buttons, **New**, **Show**, and **Copy**. This area shows the definition of the macro when it has been created.

3. Click **New** to create the macro. This inserts the default name `int Macro();` in the Name data entry field and inserts `{}` in the Macro Entry Area ready for editing.

4. Edit the default macro name so that it shows `int tutorial(var1)`.

5. Enter the macro contents to show:

   ```c
   int var1;
   {
     $printf "value=%d\n",var1$;
   }
   ```

   The Add/Edit Macros dialog looks like the example shown in Figure 9-1 on page 9-8.
Working with Macros

6. Click **Update** to pass the macro definition to RealView Debugger, where it is stored in the symbol table. This adds the new macro to the Existing Macros display list.

   If there are any errors in the macro text, you are notified when you try to pass the macro to RealView Debugger.

7. View the Output pane message:
   
   ```
   def int tutorial(var1)
   ```

8. Click **Save...** to display the Select file to save/append into dialog box where you can choose where to save the new macro definition file for later re-use.

   The macro name has been entered automatically with the extension `.inc`.

9. Save the new macro, with the default name, in your `\home` directory.

   If the chosen file already exists, RealView Debugger displays a warning message and gives you the option to append the new file or to overwrite the existing contents.

10. Click **Close** to close the Add/Edit Macros dialog box and return to the Code window.

### 9.2.2 Viewing a macro

View the contents of a macro using:

- the File Editor pane
- a standalone editor window or an external text editor
- the `SHOW` command.
Viewing a macro using any of these methods differs from viewing the macro contents in the Add/Edit Macros dialog box:

- The Macro Entry Area, in the dialog box, does not show the macro `DEFINE` command or the terminator (a period used as the first and only character on the last line).
- The `SHOW` command does not display the terminator.

Note

You cannot use the `SHOW` command to view a predefined macro.

9.2.3 Testing a macro

Test the macro you have just created by running the program and then calling the macro from the command line:

1. Click on the **Src** tab to view the source file `dhry_1.c`.
2. Set a simple breakpoint by double-clicking on line 150.
3. Click **Go** to start execution. When asked for the number of runs, use a small number, for example, 5000.
4. When the program reaches the breakpoint, enter the following command and press Enter:
   `tutorial(Int_2_Loc)`
   This displays the current value of the variable `Int_2_Loc` in the Output pane.
5. Step through the program a few more times using the macro to monitor the variable. You can use the up arrow to step back through the commands already submitted on the command line.

9.2.4 Editing a macro

You can edit the macro that you have just created and retest it to verify the changes:

1. Select **Debug → Add/Edit Debugger Macros...** to display the Add/Edit Macros dialog. The Existing Macros display list shows the `tutorial` macro you just created and it is highlighted.
2. Click **Show** to see the contents of the macro.
3. In the Macro Entry Area change the body of the macro to read:
   `$fprintf 250, "value=%d\n", var1$;`
This change displays the output of the macro in a window instead of the Output pane.

4. In the Macro Entry Area add this line at the end of the macro:
   `return(var1);`
   This change causes the macro to return the value of the variable. If the value is True, that is nonzero, then RealView Debugger continues program execution after reporting the result. If the value returned is False, that is zero, then execution stops. The macro now looks like the one shown in Figure 9-2.

   ![Figure 9-2 Editing a macro body](image)

5. Click **Update** to pass the macro definition to RealView Debugger.

6. View the Output pane message:
   `def int tutorial(var1)`

7. Click the **Save** button and save the updated macro in the same location. This generates a prompt to enable you to Append or Replace the existing file. Click **No** to replace the existing `tutorial.inc`.

8. Click the **Close** button to close the Add/Edit Macros dialog.

Test the macro you have just created by running the program and then calling the macro from the command line:

1. Select **File → Reload Image to Target** to reload the image to your debug target.

2. Enter this command:
   `>VOPEN 250`
   This opens a window ready to display the results returned from the macro.
3. Click Go to start execution. When asked for the number of runs, use a small number, for example, 5000.

4. When the program reaches the breakpoint, enter the command:
   `tutorial(Int_2_Loc)`
   on the command line of the Code window and press the Enter key. This displays the current value of the variable \texttt{Int} \_\texttt{2} \_\texttt{Loc} in the window.

5. Step through the program a few more times using the macro to monitor the variable. You can use the up arrow to step back through the commands already submitted on the command line.

### 9.2.5 Copying a macro

You can use an existing macro to form the basis of a new macro:

1. Select Debug → Add/Edit Debugger Macros... to display the Add/Edit Macros dialog and so edit the macro. The Existing Macros display list shows the tutorial macro you just created and it is highlighted.

2. Click Show to see the contents of the macro.

3. Click Copy. This automatically changes the Name: data field to show \texttt{int tutorial1(var1)}. Subsequent new macros will be called \texttt{int tutorial2..}, \texttt{int tutorial3..}, and so on. You can change the default name to your own choice.

4. In the Macro Entry Area change the body of the macro as required.

5. Click Update to pass the macro definition to RealView Debugger.

6. View the Output pane message, assuming that you do not change the default name:
   `def int tutorial1(var1)`

7. Click the Save button and save the updated macro in the usual way.

8. Click the Close button to close the Add/Edit Macros dialog.

The number of macros that can be defined is limited only by the available memory on your workstation.

### 9.2.6 Deleting a macro

With a group of macros shown in the Existing Macros display list, you can highlight selected macros and click Delete to unload them from RealView Debugger. This does not delete the files themselves if they have been saved to disk.
You can also delete a macro, and all associated symbols, using the **DELETE** command.

### 9.2.7 Calling a macro

When you first start RealView Debugger, any macros you created have been unloaded from RealView Debugger. Loading an image with all associated symbols also deletes any macros.

To load, or reload, macros into RealView Debugger:

1. Select **Debug** from the main menu to display the **Debug** menu.
2. Select **Include Commands from File**... to display the Select File to Include Commands from dialog box.
3. Highlight the required `.inc` file and then click **Open**. This loads the selected macro into RealView Debugger. If there is an error in the `.inc` file, an error message is generated in the Output pane and the macro is undefined.
4. Select **Debug → Add/Edit Debugger Macros**... to display the Add/Edit Macros dialog box where your macro is now shown in the Existing Macros display list.

You can load in several macros in this way ready for use in your debugging session. When a macro is displayed in the Add/Edit Macros dialog box, it can be changed as described previously and re-used, and resaved if required.

After a macro has been loaded into RealView Debugger, the definition is stored in the symbol table. If the symbol table is recreated, for example when an image is loaded with symbols, any macros are automatically deleted. The number of macros that can be defined is limited only by the available memory on your workstation.
9.3 Source patching with macros

When debugging your application program, errors can sometimes be temporarily patched with source statements. It is often unnecessary to edit the source code, and recompile and link. Instead, you can use a temporary patch by using breakpoint macros.

To insert a few lines of source code in your program:

1. Define a macro containing the statements that you want to insert.
2. Start a debugging session and set a breakpoint on the source line following the point where you want to insert the new lines.
3. Attach your macro to this breakpoint, see Attaching macros to breakpoints on page 4-21 for details explaining how to do this.
4. Run the program until execution stops at the breakpoint.
5. The source statements in your macro are interpreted and executed. The macro completes.
6. Program execution continues normally.

Note
Using a macro in this way might cause problems with compiler optimizations, for example the ordering of instructions might have been altered by the compiler.

You can also use a similar approach to jump over or skip lines of source code:

1. Define a macro to set the PC to a point beyond the lines that are not executed.
2. Start a debugging session and set a breakpoint on the first line to be skipped.
3. Attach your macro to this breakpoint.
4. Run the program until execution stops at the breakpoint.
5. The source statements in your macro are interpreted and executed. The macro completes.
6. Program execution continues normally from the new position of the PC.

You can also use the JUMP command for looping and skipping over commands, shown in the fragment in Example 9-1 on page 9-14. The JUMP command takes a label and an expression. If the expression evaluates to True then control jumps to the specified label. If the label is positioned earlier in the file, this loops. If the label is positioned later in the file, all intermediate commands are skipped.
The expression can test:
- symbols, using the predefined isalive() macro (see Table 9-1 on page 9-26)
- results
- local symbols, created with ADD
- file tests, using macros.

**Example 9-1 Using the JUMP command**

```plaintext
add int cnt = 20
initialise
:repeat ; /* loop 20 times */
some_commands
jump repeat,cnt ; /* repeat until cnt==0 */
;
; define some local vars if not defined.
;
; jump nodefine,isalive(cnt)==1
some_commands
:nodefine
```

For more information on the JUMP and ADD commands, see the chapter describing commands in *RealView Debugger v1.6 Command Line Reference Guide*. 
9.3.1 Patching example to reset Program Counter

The source code being debugged contains the following lines:

```c
24 count = 5;
25 for (i=0; i < MAXNUM; i++)
26 {
27     array[i]=1;
28     count=count+2;
29     k=count*i;
30 }
31
32
```

To jump over or skip lines 29 and 30, and to insert a new line temporarily incrementing count by 1:

1. Define a macro containing the statements to increment count and move the PC over the two lines:
   ```
   DEFINE patch_29()
   { count++; /* increment count by 1 */
     $SETREG @PC = #31$; /* reset program counter so skipping 29 & 30 */
     return(1); /* return 1 (True) to continue normal execution */
   }
   .
   ```

2. Start a debugging session and set an instruction breakpoint on line 29.

3. Attach your macro to this breakpoint.

4. Run the program until execution stops at the breakpoint.

5. The source statements in your macro are interpreted and executed. The macro completes.

6. Program execution continues normally.
To emulate a serial port in your source code:

1. Define a macro that emulates a serial port:

   ```c
   add unsigned long last_time; /* create local symbol */
   def int ser_port(offset,base) /* macro definition */
       int offset; /* offset of device reg */
       unsigned short *base; /* base of port */
   { /* macro definition */
     unsigned short value;
     if (offset == 0) /* control register */
       { /* control register */
         if (last_time && @cycle-last_time < 20)
           { /* control register */
             error (0, "ser_port: access less than
                 allowed time: %d", @cycle-last_time);
             return (0);
           }
         last_time = @cycle;
         value = base[offset]; /* cache written value */
         base[offset] = 0; /* reset */
         if (value == 0x20) /* want to read */
           {
             /* want to read */
           }
         } /* control register */
     } /* control register */
   } /* macro definition */
   def int ser_port(int offset,short *base) /* macro definition */
   { /* macro definition */
     if (offset == 0) /* control register */
       { /* control register */
         if (last_time && @cycle-last_time < 20)
           { /* control register */
             error (0, "ser_port: access less than
                 allowed time: %d", @cycle-last_time);
             return (0);
           }
         last_time = @cycle;
         value = base[offset]; /* cache written value */
         base[offset] = 0; /* reset */
         if (value == 0x20) /* want to read */
           {
             /* want to read */
           }
         } /* control register */
     } /* control register */
   } /* macro definition */
   ```

2. Start a debugging session and set a breakpoint on the source code to stop execution just before it accesses the serial port, for example at line 20, and attach your macro to this breakpoint:

   ```bash
   >bi \module_name\#20 ;ser_port(0,&ser_port)
   ```

3. Continue debugging using the newly-inserted serial port emulation.

   As with the previous example, this is only a temporary patch so the source code must be edited and then recompiled. Be careful, however, when using such a patch in optimized code.

### 9.3.3 Other ways to use macros

During your debugging session, you can use macros to read from or write to a file, for example:

- use the predefined `fopen()` macro to write to a specified file
- use the `FOPEN` command to create the file, followed by the `PRINTF` command to write the contents
Working with Macros

- use the predefined read macros `fgetc()` and `fread()`.
- use the predefined write macros `fputc()` and `fwrite()`.

You can also send debugging information to a window, shown in *Editing a macro* on page 9-9:

- open a window using the `VOPEN` command, or use the `WINDOW` command to see a list of available windows.
- write to the window using the predefined `error()` macro or the `PRINTF` command.

You can also use the predefined `prompt_nnn()` macros with other macros and include files. These macros enable you to interact with a user and then continue execution based on the decision, or data, entered:

- `prompt_file()`
- `prompt_list()`.
- `prompt_text()`.
- `prompt_yesno()`.
- `prompt_yesno_cancel()`.

If using these predefined macros with include files, use the `JUMP` command.
9.4 Macro language

Macros are constructed in a Kernighan and Ritchie C-like scripting language that is interpreted on the host workstation. A large set of predefined run-time macros is available, see Predefined macros on page 9-26, or you can create your own.

This section covers:
- **Macro definition**
- **Macro body** on page 9-19
- **Macro terminator** on page 9-19
- **Macro comments** on page 9-19
- **Macro local symbols** on page 9-20
- **Macro conditional statements** on page 9-20.

9.4.1 Macro definition

A macro definition must contain:
- the DEFINE command
- the macro name
- the macro body
- a terminating full stop or period (.) as the first and only character on the last line.

The syntax of a macro definition is as follows:

```c
DEFINE [return_type] macro_name([parameter_list])
[param_definitions]
{ macro_body }
.
```

where:
- **return_type** determines the type of the macro return value and is an optional component of the macro definition. The default type is `int`. For an example using return types see **Macro return values** on page 9-5.
- **parameter_list** specifies a parameter list in the same way as arguments are specified for a C function and is an optional component of the macro definition. If `parameter_list` is defined then the type must also be specified or else type `int` is assumed. The following example illustrates the use of a `parameter_list`:

```c
define int scpy(target, source)
char *target;
char *source;
```
The declaration defines arguments for the macro `scpy()`. The type of both the target and the source are declared to be pointers to a char.

### 9.4.2 Macro body

The macro body consists of the source lines of the macro and optional formal arguments.

The syntax of a macro body is as follows:

```
[local_definitions]
macro_statement;[macro_statement;]...
```

where:

- `local_definitions` defines variables used locally in the macro body.

Formal arguments can be used throughout the macro body. These arguments are later replaced by the values of the actual arguments in the macro call.

You can use debugger commands in the macro body. If used, they must be enclosed by dollar signs (`$`) and end in a semi-colon (`;`), for example:

```
$printf "value=%d\n",var1$;
```

You can also use macro arguments and local variables in RealView Debugger commands.

### 9.4.3 Macro terminator

A macro terminator is used as the last character in the macro. This is a full-stop or period (.) and is the first and only character on the last line.

### 9.4.4 Macro comments

You can use comments in your macros to document your code. Any characters identified as belonging to a comment are ignored by RealView Debugger. The following rules apply to comments in macros:

- Comments begin with a slash followed by an asterisk (`/*`) and end with an asterisk followed by a slash (`*/`).
- Line comments begin with two slashes (`//`) and end when the end of the line is reached.
- Comments in macros can be longer than one line.
Comments cannot be nested.

For example:

/* This is a comment */
// This is a line comment

9.4.5 Macro local symbols

Symbols can be created in a macro that are local to the macro. You must declare a type for macro local symbols. The type can be any legal C or C++ data type. All symbols declared within macros are automatic variables, that is the static keyword is not recognized. To create the equivalent of a global static variable, use the ADD command to create the symbol before executing the macro that references the symbol.

9.4.6 Macro conditional statements

RealView Debugger provides a set of macro flow control statements. These statements are very similar to C conditional statements and can be either simple or compound. Compound statements must be enclosed in curly braces ({}). The macro conditional statements are:

- `BREAK` on page 9-21
- `CONTINUE` on page 9-21
- `DO-WHILE` on page 9-21
- `FOR` on page 9-22
- `IF` on page 9-23
- `IF-ELSE` on page 9-24
- `RETURN` on page 9-24
- `WHILE` on page 9-25.
BREAK

The BREAK statement causes the innermost FOR...DO-WHILE loop, or WHILE loop to be exited immediately.

The BREAK statement has the following syntax:

```
break;
```

CONTINUE

The CONTINUE statement causes the remainder of the FOR...DO-WHILE loop, or WHILE loop to be ignored and the next iteration of the loop to execute.

The CONTINUE statement has the following syntax:

```
continue;
```

DO-WHILE

The DO-WHILE statement executes a given statement one or more times until an expression evaluates to False. If you have more than one statement in the DO-WHILE loop these must be enclosed in curly braces ({}).

The DO-WHILE statement has the following syntax:

```
do {
  statement; /* execute this statement */
  [statement;... /* additional statements */
} while (expression); /* while this expression is true */
```
FOR

The `FOR` statement is useful for executing a statement a given number of times. It evaluates expression_1 and then evaluates expression_2 to see if it is True, that is nonzero, or False, that is zero. If expression_2 evaluates to True, all statements are executed once.

Next expression_3 is evaluated, and expression_2 is evaluated again to see if it is True or False. If expression_2 is True, all statements are executed again and the cycle continues. If expression_2 is False, all statements are bypassed and execution continues at the next statement outside the FOR loop.

Where you have more than one statement in the FOR loop these must be enclosed by curly braces ({}).

The FOR statement has the following syntax:

```
for
  (expression_1; /* evaluate only once */
   expression_2; /* evaluate before each iteration */
   expression_3) /* evaluate after each iteration */
{
  statement; /* execute this statement */
  [statement;]... /* additional statements */
}
```

The term expression_1 can be used to initialize a variable to be used in the loop. It is evaluated once, before the first iteration of the loop. The term expression_2 determines whether to execute or terminate the loop and is evaluated before each iteration. If the term expression_2 evaluates to True, that is nonzero, the loop is executed. If expression_2 is False, that is zero, the loop is terminated. The term expression_3 can be used to increment a loop counter, and is evaluated after each iteration. This is illustrated in Example 9-2.

**Example 9-2 Using a FOR statement**

```c
j=0;
for (i = 0; i<3; i++)
  j = j+5;
k = 47;
```

In this example of the FOR statement:
- expression_1 is `i=0`
- expression_2 is `i<3`
Working with Macros

- expression_3 is i++
- statement is j = j+5
- The next statement outside the FOR loop is k = 47;

The following actions take place:
1. On entering the FOR loop, j has a value of 0, and i is set to 0.
2. Since i<3 is True, 0<3, statement is executed, and j is set to 5.
3. i is incremented, i++, and i becomes 1.
4. Since i<3 is True (1<3), statement is executed, and j is set to 10.
5. i is incremented (i++) and i becomes 2.
6. Since i<3 is true (2<3), statement is executed, and j is set to 15.
7. i is incremented (i++) and i becomes 3.
8. Since i<3 is False (3 is not less than 3), statement is not executed, and j remains 15.
9. The next statement, outside the FOR loop, is executed, that is k = 47;

IF

The IF statement is the simplest form of a macro conditional statement. It is always followed by an expression enclosed in parentheses. If the expression evaluates to zero, that is False, the statement following the expression is bypassed. If the expression evaluates to a value other than zero, that is True, the statement following the expression is executed. If you have more than one statement in the IF statement these must be enclosed in curly braces ({}).

The IF statement has the following syntax:

```c
if (expression) /* If this expression is true */
{  /* execute this statement */
  statement; /* additional statements */
  [statement;]...
}
```
IF-ELSE

The IF-ELSE statement provides a way to specify an alternative statement to execute if the IF statement evaluates to False. If the expression evaluates to True, that is nonzero, statement_1 and any following statements are executed, but statement_2 and any following statements are not executed. If the expression evaluates to False, that is zero, statement_2 and any following statements are executed, but statement_1 and any following statements are not executed. If you have more than one statement in the IF section or in the ELSE section these must be enclosed in curly braces ({}).

The syntax of the IF-ELSE statement is as follows:

```c
if (expression) /* If expression is true */
{
    statement_1; /* execute statement_1 */
    [statement;]... /* and these additional statements */
}
else /* If expression is false */
{
    statement_2; /* execute statement_2 */
    [statement;]... /* and these additional statements */
}
```

RETURN

The RETURN statement is used to return a value from a macro. The expression is evaluated, and the resulting value is returned to the caller. If a breakpoint macro returns a value of True, that is nonzero, program execution continues. If it returns a value of False, that is zero, program execution is halted. If a macro never returns a value, the macro_type must be declared as void when it is defined.

The RETURN statement has the following syntax:

```c
return [[]expression[]];
```
WHILE

The WHILE statement evaluates an expression and executes the following statement or statements until the expression evaluates to False.

The WHILE statement must be followed by an expression in parentheses. As long as the expression evaluates to True, all following statements are repeatedly executed. When the expression evaluates to False, all statements are bypassed and execution continues at the next statement outside the WHILE loop. If you have more than one statement in the loop these must be enclosed in curly braces ({}).

The syntax of the WHILE statement is as follows:

```c
while (expression) /* while this expression is true */
{
    statement; /* execute this statement */
    [statement;]... /* and these additional statements */
}
```
9.5 Predefined macros

RealView Debugger recognizes several predefined macros containing commonly used functions. Predefined macros can be used in expressions on the command line and can be called from macros that you create yourself.

Table 9-1 contains a summary of the predefined macros.

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Description</th>
<th>Syntax</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>Returns a byte value from the specified address.</td>
<td>unsigned char byte(addr) void *addr;</td>
<td>-</td>
</tr>
<tr>
<td>call</td>
<td>Performs a target function call. Use this to make an application function call from the debugger.</td>
<td>call(label, arg1,...) char *label; int arg1;</td>
<td>-</td>
</tr>
<tr>
<td>command</td>
<td>Enables you to construct a command in a string.</td>
<td>command(cmd)</td>
<td>Takes only a single argument.</td>
</tr>
<tr>
<td>dword</td>
<td>Returns a long value from the specified address.</td>
<td>unsigned long dword(addr) void *addr;</td>
<td>-</td>
</tr>
<tr>
<td>error</td>
<td>Processes error message returned from macro.</td>
<td>int error(type, message, value) int type; char *message; long value;</td>
<td>error takes a string that can have one “%d” or “%x” in it to show the value (sprintf format). The type is one of: 1=note popup 2=warning popup 3=error popup -3=fatal error popup.</td>
</tr>
<tr>
<td>fgetc</td>
<td>Returns a byte from file or window. These must be previously opened using FOPEN, VOPEN, or the predefined macro fopen.</td>
<td>int fgetc(window_ID) int window_ID;</td>
<td>-</td>
</tr>
<tr>
<td>fopen</td>
<td>Opens a file for reading, writing, or both.</td>
<td>int fopen(char *mode, int window_ID, char *file_name)</td>
<td>-</td>
</tr>
<tr>
<td>fread</td>
<td>Reads a file into a buffer.</td>
<td>unsigned long fread(void *buffer, unsigned count, unsigned size, int window_ID)</td>
<td>Acts on a window as well as a file.</td>
</tr>
</tbody>
</table>
### Table 9-1 Predefined macros (continued)

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Description</th>
<th>Syntax</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>fwrite</td>
<td>Writes a buffer to a file.</td>
<td><code>unsigned long fwrite(void *buffer, unsigned count, unsigned size, int window_ID)</code></td>
<td>Acts on a window as well as a file.</td>
</tr>
<tr>
<td>getsym</td>
<td>Takes an address and returns a local string.</td>
<td><code>char *getsym(long addr)</code>&lt;br&gt;<code>void *addr;</code></td>
<td>Example:&lt;br&gt;add char x[20]&lt;br&gt;strcpy(x, getsym(@pc))&lt;br&gt;pr x&lt;br&gt;“Start”&lt;br&gt;The PC is at the label “Start”. You can use any expression.</td>
</tr>
<tr>
<td>isalive</td>
<td>Verifies that a symbol is currently active.</td>
<td><code>int isalive(symbol_name)</code>&lt;br&gt;<code>void symbol_name;</code></td>
<td>This returns:&lt;br&gt;0=not available&lt;br&gt;1=available&lt;br&gt;2=up the stack&lt;br&gt;-1=does not exist.</td>
</tr>
<tr>
<td>memchr</td>
<td>Searches for a character in memory.</td>
<td><code>char *memchr(str1, count)</code>&lt;br&gt;<code>char *str1;&lt;br&gt;int count;</code></td>
<td>-</td>
</tr>
<tr>
<td>memclr</td>
<td>Clears memory values.</td>
<td><code>char *memclr(str1, count)</code>&lt;br&gt;<code>char *str1;&lt;br&gt;int count;</code></td>
<td>-</td>
</tr>
<tr>
<td>memcpy</td>
<td>Copies characters from memory.</td>
<td><code>char *memcpy(dest, src, count)</code>&lt;br&gt;<code>char *dest, *src;&lt;br&gt;int count;</code></td>
<td>-</td>
</tr>
<tr>
<td>memset</td>
<td>Sets the value of characters in memory.</td>
<td><code>char *memset(dest, byte_value, count)</code>&lt;br&gt;<code>char *dest;&lt;br&gt;char byte_value;&lt;br&gt;int count;</code></td>
<td>-</td>
</tr>
<tr>
<td>prompt_file</td>
<td>Displays a file containing message text.</td>
<td><code>int prompt_file(message, buff)</code>&lt;br&gt;<code>char *message&lt;br&gt;char *buff</code></td>
<td>-</td>
</tr>
</tbody>
</table>
### Table 9-1 Predefined macros (continued)

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Description</th>
<th>Syntax</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>prompt_list</td>
<td>Displays a dialog box containing message text and a choice list.</td>
<td>int prompt_text(message, buff)</td>
<td>Initially, the buffer consists of lines to show in the list (separated by \n). Example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char *message</td>
<td>strcpy(buff, “one\ntwo\nthree”)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char *buff</td>
<td>ce prompt_list(“Choose one:”, buff)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0=Cancel</td>
<td>Result is: 3 0x03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1=first list index</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2=second list index</td>
<td></td>
</tr>
<tr>
<td>prompt_text</td>
<td>Displays a dialog box containing message text.</td>
<td>int prompt_text(message, buff)</td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char *message</td>
<td>add char buff[32] = “initial”, 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char *buff</td>
<td>pr buff</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“initial”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pr prompt_text(“Enter a string”, buff)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Result is: 0 0x00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pr buff</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“new value”</td>
</tr>
<tr>
<td>prompt_yesno()</td>
<td>Displays a dialog box containing question text and buttons (Yes and No).</td>
<td>int prompt_yesno(message)</td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char *message</td>
<td>ce prompt_yesno(“Is everything OK?”)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Result is: 0 0x00</td>
</tr>
<tr>
<td>prompt_yesno_cancel()</td>
<td>Displays a dialog box containing question text and buttons (Yes, No and Cancel).</td>
<td>int prompt_yesno_cancel(message)</td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char *message</td>
<td>ce prompt_yesno_cancel(“Is everything OK?”)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Result is: 1 0x01</td>
</tr>
<tr>
<td>reg_str</td>
<td>Takes a register name from a string and returns the value.</td>
<td>unsigned long reg_str(char *name)</td>
<td>-</td>
</tr>
<tr>
<td>strcat</td>
<td>Concatenates two strings.</td>
<td>char *strcat(dest, src)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char *dest, *src;</td>
<td></td>
</tr>
<tr>
<td>strchr</td>
<td>Locates the first occurrence of a character in a string.</td>
<td>char *strchr(strl, byte_value)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char *strl;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>char byte_value;</td>
<td></td>
</tr>
</tbody>
</table>
### Table 9-1 Predefined macros (continued)

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Description</th>
<th>Syntax</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strcmp</code></td>
<td>Compares two strings.</td>
<td><code>unsigned long strcmp(str1, str2)</code></td>
<td>-</td>
</tr>
<tr>
<td><code>strcpy</code></td>
<td>Copies a string.</td>
<td><code>char *strcpy(dest, src)</code></td>
<td>-</td>
</tr>
<tr>
<td><code>stricmp</code></td>
<td>Performs string comparison without case distinction.</td>
<td><code>char *stricmp(str1, str2)</code></td>
<td>-</td>
</tr>
<tr>
<td><code>strlen</code></td>
<td>Returns string length.</td>
<td><code>unsigned long strlen(str1)</code></td>
<td>-</td>
</tr>
<tr>
<td><code>strncmp</code></td>
<td>Performs limited comparison of two strings.</td>
<td><code>char *strncmp(str1, str2, count)</code></td>
<td>-</td>
</tr>
<tr>
<td><code>until</code></td>
<td>Breaks when an expression evaluates to True.</td>
<td><code>char until(expression)</code></td>
<td>-</td>
</tr>
<tr>
<td><code>when</code></td>
<td>Breaks when an expression evaluates to True.</td>
<td><code>char when(expression)</code></td>
<td>-</td>
</tr>
<tr>
<td><code>word</code></td>
<td>Returns a word value at the specified address.</td>
<td><code>unsigned short int word(addr)</code></td>
<td>-</td>
</tr>
</tbody>
</table>
Working with Macros
This chapter explains how to use workspaces in RealView Debugger, and describes how to configure your workspace settings. Read this chapter in conjunction with Appendix A Workspace Settings Reference that contains a detailed description of the workspace settings.

This chapter contains the following sections:

- Using workspaces on page 10-2
- Viewing workspace settings on page 10-8
10.1 Using workspaces

RealView Debugger uses a workspace to define:
- connection information
- a list of projects to open when the next session starts
- debugger behavior
- windows (sizes and positions) and their attachment
- window contents and panes
- user-defined editor settings and view options.

It is not compulsory to use a workspace when working with RealView Debugger. However, using a workspace enables you to maintain persistence between debugging sessions.

Working without a workspace might be useful to debug an executable file from another developer or for compatibility with other tools. This means that some persistence details are not available.

This section describes workspaces in RealView Debugger. It includes:
- Initializing the workspace
- Workspace menu on page 10-3
- Settings options on page 10-4
- Opening workspaces on page 10-4
- Closing workspaces on page 10-5
- Projects in workspaces on page 10-6
- Creating an empty workspace on page 10-7.

For details on setting up multiple Code windows and attaching to different debug targets, see the multiprocessing chapter in RealView Debugger v1.6 Extensions User Guide.

10.1.1 Initializing the workspace

The first time you run RealView Debugger after installation, it creates a default workspace to define your initial working environment. Two files are created in your RealView Debugger home directory to store settings:

rvdebug.aws Contains workspace-specific settings that apply to the current workspace.

rvdebug.ini Contains global configuration options that apply to all workspaces, or are used when working without a workspace.
By default, at the end of your session, the .aws file is updated to save the current workspace and is used when you start your next session. The global configuration file is updated when it is edited or at the end of your session if you are working without a workspace.

**Start-up options**

You can start RealView Debugger with a specified workspace from the command line, or by using a desktop shortcut, for example:

```
rvdebug.exe -aws="C:\RealView Debugger\home\my_user_name\myws_rvdebug.aws"
```

You can start a debugging session without a workspace, for example:

```
rvdebug.exe -aws=-
```

### 10.1.2 Workspace menu

In a debugging session you can:

- edit your workspace settings and resave them ready for the next session
- set up specific workspaces containing custom settings for use during selected debugging sessions or with particular application programs
- switch between workspaces, without exiting the debugger, to continue previous debugging sessions
- close the current workspace and continue the session without a workspace.

To manage your current workspace, select **File → Workspace** from the Code window menu to display the **Workspace** menu:

**Open Workspace...**

Displays a dialog box where you can locate a workspace to open, see *Opening workspaces* on page 10-4 for details.

If you are already using a workspace, select this option to close the current workspace before the new workspace opens, see *Closing workspaces* on page 10-5 for details.

**Save Workspace**

Saves the current workspace to disk. This is useful if you have made changes since your debugging session began. The workspace file, for example rvdebug.aws, is saved with the same name and the workspace backup file is updated.
Configuring Workspace Settings

Save As Workspace...

Saves the current workspace to disk using a new name. This is useful if you have made changes since your debugging session began and want to save this new setup in a new workspace. This displays a dialog box where you can specify the new filename, for example test_workspace.aws. The newly-specified workspace becomes the current workspace.

Close Workspace

Closes the current workspace. After the workspace closes, RealView Debugger displays a list box so you can close any open objects. See Closing workspaces on page 10-5 for more details.

10.1.3 Settings options

RealView Debugger enables you to specify that settings are saved automatically at the end of the debugging session. If selected, settings are saved in your start-up file for use next time and the current workspace file is updated.

At the bottom of the Workspace menu are your current settings options:

Save Settings on Exit

Selected by default, this option saves selected user-configured settings and the current workspace when you exit RealView Debugger. This enables you to start your next debugging session in the same state.

Same Workspace on Startup

Selected by default, this option saves the current workspace pathname in your .sav file so that the same workspace is used at the next start-up.

You can unselect this option so that the current workspace is not opened by default when you next start RealView Debugger. Unless you specify a workspace on the command line, RealView Debugger then runs without a workspace.

10.1.4 Opening workspaces

If you open a new workspace, RealView Debugger adds the objects specified in the new workspace to all existing objects. This usually means that at least one more Code window opens on your desktop.

If you open a new workspace, you might see many new Code windows on your desktop. To avoid this, close open windows before opening the new workspace, see Closing workspaces on page 10-5 for details.
When you open a new workspace, it might contain settings that override the current configuration. Where there is a conflict, a warning message is displayed and the new workspace settings are used.

10.1.5 Closing workspaces

You can explicitly close your current workspace, either:

- select **File** → **Workspace** → **Close Workspace**
- select **File** → **Workspace** → **Open Workspace**... to close the current workspace before the new one opens.

If you close your current workspace, the following applies:

- The contents of the default Code window do not change.
- If there are open objects, these do not change (see Closing objects for details).
- Any open objects are saved in the workspace before it closes so that they can be re-used when it next opens.
- If there are no open objects, the current workspace closes immediately.
- If the Workspace Options window is open, this closes automatically before the current workspace closes. If you have changed any workspace settings, these are saved.
- If the Options window is open, this is not affected when the workspace closes.

Closing objects

If you close a workspace, RealView Debugger enables you to close any open objects. This might be useful to restore a clean desktop for the session, or before you open a new workspace.

After the current workspace closes, RealView Debugger displays a list selection box where you can specify which open objects you want to close, shown in Figure 10-1 on page 10-6.

**Note**

The Close Open Objects selection box is not displayed if there are no open objects.
Figure 10-1 Close Open Objects selection box

The display list shows the open objects, that is:

- connections to debug targets
- any windows open in addition to the default Code window
- open projects including user-defined projects and auto-projects (see Projects in workspaces for details).

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

**OK**  
Click this button to close selected objects and then close the selection box.

**Cancel**  
Click this button to ignore the status of any check boxes in the list and close the selection box. Use Cancel to maintain all open objects.

**Help**  
Click this button to display the online help.

You can use the Close icon to close the selection box. This is the same as clicking Cancel.

### 10.1.6 Projects in workspaces

RealView Debugger saves a project load list when the current workspace closes. This is a list of open projects maintained when the debugger starts with this workspace or when you open this workspace in a session. This list includes user-defined projects and any auto-projects where you have saved the settings.

Where you are using a project load list, you must be aware of the following when the workspace opens:

- Where a project was bound to a connection when the workspace closed, binding details are saved and this is maintained when the connection is restored.
• Project binding details saved in the workspace take precedence. This applies even where the load list opens an unbound project where you have defined a Specific_device setting.

• Where there is no connection, the order in which projects open defines the active project.

If you are already running a debugging session and you open a new workspace, the project environment might change depending on the project load list (if any). RealView Debugger forces the project binding as defined in the workspace even if this means unbinding open projects. This is true even if the open projects include an autobound project, that is a project where you have defined a Specific_device setting.

If the workspace opens with no saved binding details, the current project environment does not change. This is true even if the project load list contains a project where you have defined a Specific_device setting.

**Note**

There is no warning when the project environment changes as a result of opening a workspace into a debugging session.

If you are licensed to work with multiple projects in multiprocessor debugging mode, the workspace restores the project environment based on:

• your connections
• the order in which projects open
• saved project binding details
• open windows and their attachment.

See Chapter 11 *Managing Projects* for full details on working with projects and project binding.

### 10.1.7 Creating an empty workspace

You can create a blank workspace settings file at any point during your debugging session. To do this select **File → New → Workspace**... from the Code window main menu. This displays the New Workspace dialog box.

Use this to create an empty file, for example New_workspace.aws, in the new location. This becomes the current workspace. You must save settings to this new workspace settings file if you want it to be available at the next start-up.

If you are already working with a workspace this closes and then the new workspace opens ready for you to use. This does not override the current configuration.
10.2 Viewing workspace settings

RealView Debugger provides the Workspace Options window to enable you to examine, and change, workspace settings. Use this interface to see the contents of the .aws file and the .ini file.

There are descriptions of the general layout and controls of the RealView Debugger Settings windows in the RealView Debugger online help topic Changing Settings. This chapter assumes familiarity with the procedures documented in that topic.

This section contains:
- Using Settings windows
- Options window
- Workspace Options window on page 10-9
- Groups and settings on page 10-11.

10.2.1 Using Settings windows

Select Tools from the Code window main menu to display the Tools menu and access your current workspace settings or global configuration options:

Workspace Options...
Displays the Workspace Options window where you can view the current workspace settings or make changes.

Options...
Displays the Options window where you can view the global configuration options, used by the current workspace, or make changes. You have access to the global options when you are working without a workspace.

10.2.2 Options window

The Options window enables you to examine, and change, your current global configuration options. These settings are saved in the file rvdebug.ini and are included when the default workspace opens for the first time.

If you are working without a workspace, use this window to make the changes described in the rest of this chapter.
10.2.3 Workspace Options window

The Workspace Options window enables you to examine your current workspace settings and edit these settings to change the workspace or to create your own workspace files. The first time RealView Debugger opens the default workspace file, rvdebug.aws, the Workspace Options window contains only the start-up settings, shown in Figure 10-2.

![Figure 10-2 Workspace Options window](image)

The main interface components of this window are:

**Main menu**  This contains:

- **File**  Displays the File menu where you can save the workspace file after you have made changes.
- **View**  Displays the View menu to toggle the display to show all the settings or only those that have been edited.
- **Help**  Displays the online Help menu.

**Save icon**  Click this icon to save the workspace settings file to disk. The name of the current file is shown as the first entry in the left pane.

**Description**  This field displays a one-line description about an entry selected in the panes below.
List of Entries and Settings Values

The left pane of the Workspace Options window, the List of Entries pane, shows workspace entries as a hierarchical tree with node controls. Groups of settings are associated with an icon to explain their function:

- **Red disk**  This is a container disk file, for example to specify an include file.
- **Yellow folder**  This is a parent group containing other groups (rules pages) and/or entries.
- **Rules page**  A rules page is a container for settings values that you can change in the right pane. This icon only appears in the left pane.

When you close down RealView Debugger, your workspace settings file is updated with the current configuration, for example projects, connections, and open windows. An asterisk (*) is placed at the front of an entry to show that it has changed from the default or was created by RealView Debugger. An example settings file is shown in Figure 10-3.

![Figure 10-3 Example Workspace Options](image)

If you click on an entry in the left pane, a red box is drawn around it and the Description field is updated. At the same time, the right pane, the Settings Values pane, is updated to show the contents of the highlighted group.
10.2.4 Groups and settings

Workspace settings, in the left pane, are grouped according to their function:

**Workspace file**

This is the current workspace settings file. Click on this entry to see the full pathname in the Description field.

**Global configuration file**

This is the global configuration file, `rvdebug.ini`, included in the current workspace.

You can set up DEBUGGER, CODE, or ALL groups in your workspace settings file or in your global configuration file. RealView Debugger issues a warning if conflicts are detected when the workspace opens and uses settings from the new workspace file.

**DEBUGGER**

These settings govern the behavior of generic actions in the debugger. These controls are used in conjunction with other processor-specific controls.

**CODE**

These settings govern the behavior of all Code windows. They control the display characteristics of windows, including size and position, and any user-defined buttons created on the toolbars (not available in this release).

**ALL**

These settings govern the behavior of the editor, the editor display, and access to source code. The settings in the ALL group are used in conjunction with the settings in the DEBUGGER group and the CODE group and might be overridden by settings in either of these two groups.

**PROJECTS**

This specifies a project load list, that is a project, or projects, opened when the debugger starts with this workspace or when you open this workspace in a session. This list includes user-defined projects and any auto-projects where you have saved the settings.

This group is created automatically when RealView Debugger closes down with open projects, or if you close the current workspace.

**WINDOW**

This is a special group of windows internals maintained by RealView Debugger. An entry is created for each open window. Entries cannot be edited.

You must not delete these entries.
**CONNECT**  When you close down, you have the option to save connection details so that the same connections are used when RealView Debugger starts up with the same workspace.

This is a special group, to specify connections, maintained by RealView Debugger. An entry is created for each connection. Entries cannot be edited.

You must not delete these entries.

For a full description of all the workspace settings you can change see Appendix A *Workspace Settings Reference*. 
10.3 Configuring workspace settings

The following notes apply to changing your workspace settings:

- Settings are applied in the order they are shown in the settings hierarchy in the left pane. This means that settings in the workspace file take priority over global configuration settings if a conflict arises when you open a workspace.

- If you edit the workspace settings, the .aws file is updated when you save the change. This change takes effect in any new Code windows you open in the current session.

- Use the Options window to make changes to global configuration options saved in the rvdebug.ini file.

- If you edit the global configuration options, the .ini file is updated when you save the workspace file. This change takes effect when the workspace next opens.

- Do not change the same setting in the Workspace Options window and the Options window at the same time because the views might not be consistent.

This section describes:

- Changing settings
- Copying entries on page 10-14
- Pasting entries on page 10-15
- Cutting entries on page 10-17
- Resetting entries on page 10-17.

10.3.1 Changing settings

This section includes examples of changes that you can make to your current workspace. Select Tools → Workspace Options... to display the Workspace Options window to edit the workspace file. This means that changes take effect in the current workspace:

- Configuring the Code window
- Setting up debugger options on page 10-14.

When you have saved a change, select View → New Code Window to display a new Code window to see the effect.

Configuring the Code window

To change the size of the Code window:

1. Expand the CODE group.
Configuring Workspace Settings

2. Expand the Pos_size group.
3. Right-click on the default Num_lines setting.
4. Select **Edit Value** from the context menu.
5. Use in-place editing to set the value to 0x040.
6. Press Enter to confirm your setting.
7. Save the updated version of the workspace settings file.

**Setting up debugger options**

To change the height of the Output pane:

1. Expand the **DEBUGGER** group.
2. Expand the **Command** group.
3. Right-click on the default **Num_lines** setting.
4. Select **Edit Value** from the context menu.
5. Use in-place editing to set the value to 10.
6. Press Enter to confirm your setting.
7. Save the updated version of the workspace settings file.

___ **Note**

To restore the Code window, select the option **Reset to Empty**.

10.3.2 Copying entries

When you are working in the Workspace Options window, context menus include options to enable you to copy settings so that you can make changes quickly. The options that are available depend on the:

- pane you are working in
- contents of the clipboard
- relationship between what is on the clipboard and the entry under the cursor when you right-click.
The available options are:

**Copy**
Select this option to make a copy of an entry to the clipboard ready for pasting.
This option is always available in the left pane to copy settings groups.
When you are working in the right pane, this option depends on the chosen entry.

**Make Copy...**
Where permitted, this option enables you to make a copy of the chosen group. A dialog box enables you to define a new name for the copy.

### 10.3.3 Pasting entries

When you are working in the Workspace Options window, context menus include options to enable you to paste settings so that you can make changes quickly. Like the copy options, the paste options that are available depend on the:

- pane you are working in
- contents of the clipboard
- relationship between what is on the clipboard and the entry under the cursor when you right-click.

The available options are:

**Paste Group Into**
This option is only available if you right-click on a settings group, or a container disk file, with a settings group already copied to the clipboard.
This option is usually available in the left pane to paste settings groups into the settings file, or to copy between the workspace settings file and the global configuration file.
When you are working in the right pane, this option depends on the chosen entry. This means that you might not be able to paste the contents of the clipboard into the chosen location.

**Paste Rule Here**
Certain settings in the right pane are classed as rules. In particular, you can use rules to specify settings for projects when you are working in the Project Properties window (see Configuring project properties on page 11-52 for details).
This option is only available if you right-click on a settings group, or a container disk file, with a single rule setting already copied to the clipboard.

This option is available in the left or right pane if you select a settings group that can accept the rule currently on the clipboard. This means that you might not be able to paste the contents of the clipboard into the chosen location.

**Paste as 1st Child**

To see this option, you must right-click on a parent group with a child group already copied to the clipboard.

Use this option, in the left pane, to paste a settings group into a parent group, that is to create a sibling group.

When you are working in the right pane, use this option to paste a child group. The paste only succeeds if the chosen parent group can accept the child. This means that you might not be able to paste the contents of the clipboard into the chosen location.

This option might be replaced by **Paste After**. This depends on the chosen entry.

**Paste After**

Use this option, like **Paste as 1st Child**, to paste a settings group into a parent group, that is to create a sibling group.

This option enables you to specify the relationship between the new group and other siblings. This determines the order in which settings are used.

**Paste String**

This option is only available in the right pane if you right-click on a settings group, or a container disk file, with a string setting already copied to the clipboard.

The paste only succeeds if you select a setting that can accept the string currently on the clipboard. This means that you might not be able to paste the contents of the clipboard into the chosen location.

**Paste Value**

This option is only available in the right pane if you right-click on a settings group, or a container disk file, with a value setting already copied to the clipboard.

The paste only succeeds if you select a setting that can accept the value currently on the clipboard. This means that you might not be able to paste the contents of the clipboard into the chosen location.
10.3.4 Cutting entries

When you are working in the Workspace Options window, context menus include the option **Cut** to enable you to mark an entry for deletion. The entry is grayed out until you paste it into a new location.

If you cut another entry before you paste the first entry, the first entry is restored.

If you cut an entry, do not delete it until it has been pasted. If you delete the cut entry, it is no longer available to paste. This also applies to entries that you copy.

Always use **Delete** to remove unwanted entries.

10.3.5 Resetting entries

An asterisk is placed at the front of any entry that you edit in the workspace settings file to show that it has changed from the default. This also applies to entries maintained by RealView Debugger. You can reset all values using the **File → Reset** option from the window menu. This updates the window with the settings currently saved on disk. You are warned that any changes made since you last saved will be lost.

When you have changed a value, right-click to see the context menu showing the option **Reset to Default**. Select this to change the value to the default and cancel any changes.

Right-click on a group of settings and select **Delete Contents** from the context menu to reset it back to empty. This deletes all the changed settings and restores the defaults. There is no undo.
Chapter 11
Managing Projects

This chapter describes in detail the features of RealView Debugger that help you to manage software projects. It includes the following sections:

- About managing projects on page 11-2
- Using the Project and Tools menus on page 11-12
- Managing your build tools on page 11-18
- Managing user-defined projects on page 11-21
- Creating a new user-defined project on page 11-29
- Building your application on page 11-41
- Managing project properties on page 11-46
- Configuring project properties on page 11-52
- Managing build target configurations on page 11-72
- Using the Project Control dialog box on page 11-81
- Managing projects in the Process Control pane on page 11-84
- Project binding on page 11-90
- Managing multiple projects on page 11-102.
11.1 About managing projects

Setting up a project in RealView Debugger is not required for debugging, but it can provide an aid to development. RealView Debugger uses projects to save your list of files, understand your build model, and maintain a record of your project-level preferences. A project also enables RealView Debugger to save, and load automatically, specified debugging states, for example breakpoints.

A user-defined project can also speed up your debugging session because your project stores information and settings about your image that can then be used to help locate errors, and rebuild your source code into an executable image or program.

This section gives an overview of projects in RealView Debugger. It includes the following sections:

- What is a project?
- User-defined projects on page 11-3
- Auto-projects on page 11-5
- Organizing user-defined projects on page 11-5
- Build tools for user-defined projects on page 11-7
- Project properties on page 11-7
- Build target configurations on page 11-7
- Makefiles on page 11-8
- Project binding on page 11-9
- Projects in the workspace on page 11-11
- Projects in Connection Properties on page 11-11.

11.1.1 What is a project?

A project is the highest level structural element that you can use to organize your source files and determine their output. The project information and settings can be managed by RealView Debugger.

You can:

- create a range of software projects using predefined templates included in the root installation
- access image-related settings through auto-projects
- define build tools to use with all projects or only a subset
- view and change project properties without leaving the Code window
- define different build target configurations
Managing Projects

• associate projects with connections or specific target processors
• set up a project environment automatically when the workspace opens
• open projects automatically when you connect to a specified debug target.

Types of project

There are two types of project in RealView Debugger:

User-defined projects

Projects that you create and set up. See User-defined projects for more details.

Note

If you have a user-defined project, do not load the image without first opening the project. If you do, RealView Debugger creates an auto-project, and does not load the settings from your project settings file.

Auto-projects

Projects that RealView Debugger uses automatically when you load an image directly. RealView Debugger defines project properties based on the image contents. An auto-project is a no-build project because RealView Debugger is unable to determine the build model from the image. See Auto-projects on page 11-5 for more details

11.1.2 User-defined projects

A user-defined project is set up and managed by you. If you have existing source files, RealView Debugger enables you to select the source files during project creation, or add them later.

The project information and settings are stored in a project settings file. This has the same name as the project and an extension of .prj. The file is placed in the project directory that you specify when you create a project. For example, std_proj_1.prj is the project settings file for the project named std_proj_1.

A user-defined project defines:
• a list of source files
• build rules, or specifies no build at all
• build rules including compiler, assembler, and linker
• custom build rules
• build target configurations such as debug, debug/release, and release
Managing Projects

- makefiles to use
- runtime settings such as the image name and loading rules
- image-specifics such as semihosting on and vector-catch enabled.

**Note**

If you have created a user-defined project, it is recommended that you open this first to load and debug the associated image, or images. This avoids the creation of an auto-project and enables you to save any new settings or changes to the build model.

**Types of user-defined project**

The types of user-defined project you can create and manage in RealView Debugger are:

**Standard**  A Standard project is composed of sources to compile, and/or assemble, and link. A Standard project might also contain a custom build model. RealView Debugger creates the project makefile automatically.

**Library**  A Library project enables you to compile and/or assemble files to put into a library. RealView Debugger creates the project makefile automatically.

**Custom**  A Custom project enables you to specify your own makefile, or to specify an external build program or script, or to use no build model.

**Container**  A Container project contains other projects and can be used to:
- share components within, and between, development teams
- divide up complex builds into libraries
- contain projects for different processors.

**Note**

By default, if you open a Container project, you are working on multiple projects. You can perform additional operations on the individual projects in a Container project.

**Copy**  You can copy existing user-defined projects to try variations of your application program or to test different development environments.

**Note**

Regardless of the type of project you create, project files can be controlled by your version control software if required. See Chapter 14 *Working with Version Control Systems* for details on how to do this.
11.3 Auto-projects

When an image, for example `test_image.axf`, is loaded to a debug target, RealView Debugger checks to see if an auto-project with the same name (`test_image.axf.apr`) exists in the same location. If an auto-project exists, RealView Debugger opens this to provide certain project settings for the current session. If no auto-project exists, RealView Debugger creates an in-memory auto-project to use in this session by reading settings directly from the image.

--- Note ---
When RealView Debugger creates an auto-project, it derives a project name from the name of the associated image. It is this project name, and not the image name, that is shown in the title bar of the default Code window. However, only the image name is visible in the Process Control pane.

---

Auto-projects are used to store configuration information for the image, including:
- semihosting
- vector catching
- automatic breakpoint setting
- start-up and load commands.

The `.apr` file has an identical format to that of a user-defined Custom project settings file. There is no build model for the current image because this was unknown when the image was loaded. The Process tab in the Process Control pane contains a Settings group that indicates whether or not you have saved the auto-project settings to a `.apr` file.

--- Note ---
You can merge the settings from an auto-project when you create a new user-defined project. See Merging auto-project settings into a project on page 11-36 for details on how to do this.

---

See Working with auto-projects on page 11-85, for details on how to manage auto-projects.

11.4 Organizing user-defined projects

A user-defined project is a collection of source files, library files, and other input files. You can organize the files in a project in different ways to provide a logical structure to your source.
Managing Projects

It is recommended that you decide the best way to organize your user-defined project files before using the RealView Debugger project management options. How projects are organized affects the extent to which files can be shared between developers.

When you create a user-defined project, you specify the project name and the project base directory (see Creating a new user-defined project on page 11-29). The source files used in the project do not have to be in the project base directory but can be located elsewhere and are referred to using relative pathnames, where possible.

RealView Debugger generates a warning message when projects are not self-contained so that you can decide to cancel an operation or continue. In general, keeping source files together within the project base directory, and any subdirectories, is the preferred option for single-user projects.

When working with projects and making changes, additional files are created, for example safety backup files. See the tutorial in RealView Debugger v1.6 Essentials Guide for details of these files.

Project size

There is no limit to the number of files that RealView Debugger can handle in a single project. However, your operating system might impose a limit on the number of files that can be passed to the linker. Keeping all your source files in the project base directory, and using short filenames, can help to maximize the number of files in a project. However, using libraries is the recommended approach for large projects.

There is a limit imposed on the line length in the generated makefile. This is defined by the setting MAXLINELENGTH set to 32768, in the file \etc\startup.mk. You can make this longer by editing the file.

Deleting user-defined projects

In general, you do not have to delete user-defined project files or the contents of a project base directory. After you have created a user-defined project, you can add and delete files as necessary using the project management options from the default Code window. You can also change your build model and other project components using the Project Properties window.

Deleting user-defined project files is not recommended where projects are not single-user, self-contained projects as this might prevent other developers from accessing your source files, built files, or build model.
11.1.5 Build tools for user-defined projects

RealView Debugger provides support for multiple toolchains in user-defined projects. The project toolchain defines the base settings for development tools, for example ADS. When you have specified these settings they are used for every project that you build using that toolchain. You can override these settings in your project so that different tools are used for a project or for a particular set of files forming part of a project.

Depending on your installation, RealView Debugger can locate toolchains for projects using environment variables already set, for example RealView Compilation Tools.

It is recommended that you define the location of your build tools before creating your first project. If you do not do this, RealView Debugger asks you to specify it when saving your first project. See Managing your build tools on page 11-18 for full details.

11.1.6 Project properties

The Project Properties window enables you to view the project settings for user-defined projects or auto-projects. You can use this to change user-defined project options as build configurations change, or to build an executable image without having to manually edit any files used by RealView Debugger. You are recommended to use this interface because this populates the interrelated files that control the build process and constructs the required makefiles.

However, you can choose not to use this interface and to set up your own commands and scripts if required.

If you are using an auto-project, you can use this interface to define image load options that are stored with the current image and used the next time the image is loaded.

See Managing project properties on page 11-46 and Configuring project properties on page 11-52 for details on how to manage and configure project properties.

For a full description of all project properties and available settings, see Appendix B Project Properties Reference.

11.1.7 Build target configurations

The most important element in a user-defined project is the build target configuration. This defines how the source files within your project are processed, not the project itself, and enables you to build the same image in different ways. A build target configuration is, therefore, a specific arrangement of build options that are applied to all, or some, of the source files in a project to produce an output file, such as an executable image, library, or code listing.
--- Note ---

A build target configuration is distinct from a debug target, such as an ARM development board. The way to configure your debug targets, and to add new targets, is described in RealView Debugger v1.6 Target Configuration Guide.

A user-defined project defines at least one build target configuration, for example a Debug build, or a Release build. RealView Debugger defines three build target configurations:

- **Debug**  
  This builds output files that you can fully debug, at the expense of optimization. This provides the best debug view while you are developing your code.

- **DebugRel**  
  This builds output files that provide adequate optimization and give a good debug view.

- **Release**  
  This builds output files that are fully optimized, at the expense of debug information.

You can define a specific build order for the build target configurations in a project. Build target configurations can share files in the same project, while using their own build settings. The Project Properties window enables you to define and set up such relationships.

Each build target configuration has a corresponding directory where the built files are placed. The directories have the same name as the build target configuration, and are subdirectories of your top-level project directory.

For details on build models and build target configurations, see Managing build target configurations on page 11-72.

### 11.1.8 Makefiles

Makefiles are used to describe the relationships between files in your application program and provide commands for building these files. They provide a database of rules detailing information about files and specifying how they are used in a build. Makefiles are used, therefore, to automate the build process. Projects can wrap user-defined makefiles, or you can edit the RealView Debugger templates to customize your build process, for example to use specific source control.

**Standard and Library project makefiles**

Creating a Standard or Library project means that makefiles are built for you using the RealView Debugger templates.
When you configure and save your project settings file, RealView Debugger uses a template file named `gen_***.mk` to generate the required makefile for your project. Different templates are installed depending on the installation type, that is Custom or Typical, and the licenses you possess.

The template name is chosen based on the toolchain you specify when you create the project, that is the target processor and build tools. For example, the template `gen_arx.mk` is used for building executable files and libraries with ARM compilation tools to run on the ARM family of processors. This template provides the ARM-ADS specific makefile layouts for the `genmake` utility used by RealView Debugger.

You can copy and edit the template file to add details specific to your requirements. If you do this, you can then specify the new template file in your project `BUILD` group.

The file `\etc\startup.mk` is used by the `make` utility when working with projects on Windows to define default settings.

A makefile is created in your project directory for each build target configuration. The following makefiles are created for the default build target configurations, where `projectname` is the name of your project:

- `projectname_Debug.mk`
- `projectname_DebugRel.mk`
- `projectname_Release.mk`.

**Custom project makefiles**

Creating a Custom project means that you can use your own makefiles. RealView Debugger does not automatically generate makefiles for a Custom project.

See *Steps for creating a Custom project* on page 11-32 for instructions on creating Custom projects.

---

**Note**

When you create a Custom project, RealView Debugger specifies a default make command, that includes the control character `$e`. To successfully build your Custom project, remove the `$e` control character, and use your own arguments as required. See *Using your own make command* on page B-79.

---

### 11.1.9 Project binding

When you create a user-defined project, you define a toolchain associated with the target application, for example `ARM-ADS`. If you are connected to a target and you open this project, RealView Debugger tries to bind the project to the connection using this
default binding. If successful, the project name is enclosed in round brackets in the default Code window title bar. The Process Control pane (if visible) is updated to show details of the current process and enables you to access the project properties.

If you open a project and there is no connection to bind to, the project is unbound. In this case, the project name is enclosed in angled brackets when it is displayed in the Code window title bar. Although, it is not visible in the Process Control pane, you can still access the project properties from the Project menu.

A connection can only have one project bind to it at any one time. If you open a project with a project already bound to your connection, RealView Debugger asks for confirmation to use default binding to bind the new project.

--- Note ---
If you are licensed to work in multiprocessor debugging mode, default binding enables a project to bind to all connections.

---

When a project binds to a connection selected actions can be carried out on that connection, for example image loading actions, or RealView Debugger can execute commands saved in the project settings file.

When you are working with multiple projects, binding enables you to access your image details, view the project properties, and make changes to the project quickly.

When you load an image directly to a target, the auto-project (associated with the image) binds to the connection by default.

You can change binding manually using the Project Control dialog box, see Using the Project Control dialog box on page 11-81 for details.

See Project binding on page 11-90 for full details on project binding.

**Autobinding**

You can define how RealView Debugger binds a project using *autobinding*. You can specify exactly the processor (or processors) to bind to and so restrict your project to specified debug targets. *Autobound* projects take precedence when RealView Debugger tries to bind, depending on the project environment. See Autobinding on page 11-91 for full details.
11.1.10 Projects in the workspace

RealView Debugger saves a project load list when the current workspace closes. This is a list of open projects maintained when the debugger starts with this workspace or when you open this workspace in a session. This list includes user-defined projects and any auto-projects where you have saved the settings.

See Projects in workspaces on page 10-6 for more details.

11.1.11 Projects in Connection Properties

You can configure a debug target, using the Connection Properties window, so that one or more projects open automatically as soon as you connect to that target. The projects can be user-defined projects or auto-projects.

See the chapter describing configuring custom targets in RealView Debugger v1.6 Target Configuration Guide for more details.
11.2 Using the Project and Tools menus

The Project and Tools menus of the default Code window main menu provide options to create user-defined projects, manage projects and project properties, and build applications. The Actions toolbar also contains quick buttons for access to some of the menu options.

This section describes these project management features:

- **Active project**
- **Using the Project menu**
- **Using the Tools menu** on page 11-15.

### 11.2.1 Active project

You can have several projects open at any time. The Project and Tools menus offer project-level commands that apply to a single project. If you select a command from these menus, RealView Debugger applies the chosen operation to the active project.

In a debugging session, the active project is usually the last project you open. If you are connected to a debug target, then the active project is the last project that RealView Debugger opens and successfully binds to the connection. The active project is shown in the default Code window title bar.

---

**Note**

When working with multiple projects, Container projects, or in multiprocessor debugging mode, you might have to change the active project so that you can access the project properties, and perform project-level operations, from the Code window. See Changing the active project on page 11-104 for details on how to do this.

---

### 11.2.2 Using the Project menu

The Project menu of the Code window contains project control commands to create projects, redefine project settings, and control access to projects during your debugging session. Some options in the Project menu are not available unless at least one project is open or a source file is visible in the File Editor pane.

The options available are:

**New Project...**

Displays the Create New Project dialog box where you can create a new user-defined project. See Common steps for creating a user-defined project on page 11-29 for details.
Open Project...
Displays the Select Project to Open dialog box where you can locate a user-defined project to open. A user-defined project is saved with the .prj extension.

You can also open a user-defined project by dragging the appropriate project settings file, with the .prj extension, from Windows Explorer and dropping it into the File Editor pane.

You can also open a project from the Recent Projects list.

Project Control...
This option displays the Project Control dialog box where you specify actions to perform on one or more open projects. See Using the Project Control dialog box on page 11-81 for details.

Close Project...
Closes an open project. When two or more projects are open, a list selection box is displayed for you to choose which project or projects to close.

Recent Projects
Displays a list of up to nine user-defined projects that you have used in this or previous debugging sessions. On first opening RealView Debugger, this option is grayed out. Each project you create is added to this list, and the list is saved in your start-up file.

Project Properties...
Displays the Project Properties window where you can view and change project settings.

The project properties are saved in the project settings file (*.prj or *.axf.apr). The entries are populated when you first create the project, but you can amend them as required. If you load an image directly, you can use this option to access project properties for the associated auto-project, created by RealView Debugger. See Managing project properties on page 11-46 for details.

Build-Tool Properties...
Displays the Build-Tool Properties window to enable you to specify the location of development tools accessible to RealView Debugger for building. This option is not used for auto-projects.

It is recommended that you define the build tools before creating your first project. See Managing your build tools on page 11-18 for details.
Add This File to Project

Adds the source file that is currently selected in the File Editor pane to the active user-defined project. This option is enabled when a source file is selected in the File Editor pane and a user-defined Standard or Library project is open.

For Custom projects, you must edit your own makefile to add files.

Add Files to Project...

Displays the Select file(s) to Add dialog box to locate one or more source files to add to the active user-defined project. You can also select files from your personal favorites list. This option is enabled when one or more user-defined Standard or Library projects are open.

For Custom projects, you must edit your own makefile to add files.

Update Dependencies

Updates dependencies for the active user-defined project. This option is enabled when a user-defined project is open that contains a makefile.

View Project Source files...

Displays the Project Source View selection box. The display list shows those source files defined in the makefile for the active user-defined project. Select a source file from the list and click Edit to open the file in the File Editor pane. This option is enabled when a user-defined project is open that contains a makefile.

You can also view source files for a project using the Process Control pane. See Managing projects in the Process Control pane on page 11-84 for details.

If you select certain options from the Project menu, for example Add Files to Project... or Update Dependencies, the project makefile is regenerated. When this happens, the Output pane switches to the Build tab and displays details about the makefile creation process, shown in Figure 11-1 on page 11-15.
Do not try to add any files to your project until this process is complete and the results are visible in the Output pane.

See Managing user-defined projects on page 11-21 for examples using the Project menu.

11.2.3 Using the Tools menu

The Tools menu contains commands that control your build processes and provide assistance when locating and correcting errors during a build. See Tools menu build options for a summary of the build options available from this menu.

Other nonbuild related options are available from this menu. See Other Tools menu options on page 11-16 for a summary of these options.

Tools menu build options

The Tools menu offers the following build options:

Build... Rebuilds an executable image or library.

Selecting this option without an open project, or where the active project is an auto-project, displays the Build dialog box populated with image settings, if available.

You can also rebuild executable or library files from RealView Debugger without making a connection using this option.

See Using the Build dialog box on page 11-44 for details.

Build This File

Creates an object file from the current source file selected in the File Editor pane. With a user-defined project open, the stored build model is used to perform the rebuild for the project.
If you select this option without an open project, RealView Debugger displays the Build dialog box populated with object settings. You can also use this option to rebuild a source file from RealView Debugger without making a connection. This option is enabled when a file is selected in the File Editor pane. See Using the Build dialog box on page 11-44 for details.

**Next Line/Error**

If rebuilding generates a list of errors, this option enables you to move through the errors, make corrections, and check your code. See Finding build errors on page 11-45 for details. This option is also used when searching for text in files as part of your development or debugging session. Select this option to move to the next matching occurrence of the search string or expression as displayed in the FileFind tab of the Output pane. See Chapter 13 Searching and Replacing Text for details.

**Stop Build/Find**

Stops any build, rebuild, or search operation that is in progress.

**Clean (remove objects)**

Removes object and executable files for a project. You can use this to remove any files built by the project. This option is enabled when a user-defined project is open that contains a makefile.

**Rebuild All (Clean+Build)**

Removes object, executable, or library files for a project, and then rebuilds the project using the stored build model. Use this option to force a build from scratch. This option is enabled when a user-defined project is open that contains a makefile.

**Other Tools menu options**

Other Tools menu options are available, and are described in detail in other parts of the RealView Debugger documentation:

**New Editor**

Displays options to use standalone editor windows to give you access to all the built-in editing features. See Chapter 12 Editing Source Code for more details.
Analyzer/Trace Control

Displays options to configure and display analyzer and trace information, and to connect to a logic analyzer. This option is available only if you have a Trace license. See the chapter describing tracing in RealView Debugger v1.6 Extensions User Guide for more details.

Simulation Control

This option is not available in this release.

Workspace Options...

Displays the Workspace Options window. See Viewing workspace settings on page 10-8 for details on configuring your workspace.

Options...

Displays the Options window, where you specify global configuration options. See Viewing workspace settings on page 10-8 for details on configuring global options.

Using the Actions toolbar

The Actions toolbar contains buttons that replicate selected build options from the Tools menu:

- **Build**
  - Rebuilds an executable image or library file. See the description of the Build... option in Tools menu build options on page 11-15.

- **Compile**
  - Creates an object file from the current source file selected in the File Editor pane. See the description of the Build This File option in Tools menu build options on page 11-15.

- **Stop Compile/Build/Find**
  - Stops any build, rebuild, or search operation that is in progress. See the description of the Stop Build/Find option in Tools menu build options on page 11-15.
11.3 Managing your build tools

When you install RealView Debugger for the first time, it determines the location of your build tools to use for all user-defined projects that you create. However, you can override this setting or specify a different tool for selected projects.

Note

Auto-projects do not contain a build model and so make no use of build tools. This section applies to user-defined projects only.

If you have more than one version of the ARM build tools, for example ADS v1.2 and RVCT v2.0, RealView Debugger uses the latest version.

Note

If you have upgraded your build tools since you created a project, RealView Debugger prompts you to upgrade when you next open that project. See Upgrading the project toolchain on page 11-25 for more details.

If you are not using a project but you want to rebuild a particular file, or set of files, RealView Debugger gives you the option to use the default build tools or to specify your own makefile or build script.

The rest of this section describes how to access your build tools:

- Build-Tool Properties
- Defining build tools on page 11-19.

11.3.1 Build-Tool Properties

Select Project → Build-Tool Properties... from the default Code window to display the Build-Tool Properties window, shown in Figure 11-2 on page 11-19.
Managing Projects

Figure 11-2 Build-Tool Properties window

The Build-Tool Properties window enables you to view the current build tools and to edit these settings to specify other tools. The entries displayed in this window depend on the type of installation and the licenses that you have.

Genmake location file

The genmake location file, genmake.loc, makes the link between the project properties and the toolchain settings to use. For example, if you specify a particular compiler option, this file ensures that the correct option is used for your toolchain to get the result that you want.

The default file is stored in \etc. When the debugger runs for the first time, RealView Debugger copies this file into your home directory. From that point on, RealView Debugger automatically looks for this file in your home directory at each start-up. The search expands to other directories if this file is missing. This file is prebuilt with the appropriate entries for using RealView Debugger with your licensed target processors.

In the example shown in Figure 11-2, RealView Debugger has automatically located the ADS v1.2 build tools for ARM-based projects.

11.3.2 Defining build tools

If RealView Debugger is unable to determine the path to your build tools, or you want to override this setting, you must define the tools used to build your executable files.

To specify a build tool:

1. Select Project → Build-Tool Properties... to display the Build-Tool Properties window.
2. Select the `PROC=` group that you want to change in the List of Entries pane, the left pane.
   For example, click on `PROC=ARM_ADS` if you have ADS. This node is expanded so that the contents are visible in the Settings Values pane, shown in Figure 11-2 on page 11-19.

3. Right-click on the setting you want to change, for example Compiler, in the Settings Values pane, the right pane.

4. Select **Edit as Filename** from the context menu to display the Enter New Filename dialog box. Locate the compiler that you want to use.
   If you know the full pathname, select **Edit Value** from the context menu, and enter the required pathname.
   Click **Save** to confirm the setting.

5. Select **File → Save and Close** to close the Build-Tool Properties window and update the `genmake.loc` file.

**Working with open projects**

If you change build tools with user-defined projects open, closing the Build-Tool Properties window displays a selection box where you can specify which projects are updated.

Select the projects to which RealView Debugger must apply the new setting and click **OK**. Click **Cancel** to leave the projects unchanged. RealView Debugger regenerates the makefile(s) for the chosen project(s). Wait for this to complete before performing any more project operations.

---

**Note**
The selection box shows all open projects and, therefore, might contain projects where the new build tool is not applicable.
11.4 Managing user-defined projects

This section introduces the specific tasks and information you require to manage user-defined projects. It contains:

- Opening a user-defined project
- Generated makefiles on page 11-22
- Project properties on page 11-23
- Adding files to a user-defined project on page 11-23
- Upgrading the project toolchain on page 11-25
- Closing a user-defined project on page 11-27.

11.4.1 Opening a user-defined project

To open an existing user-defined project:

1. Start RealView Debugger.
2. Connect to the ARM7TDMI core using ARMulator.
3. Select Project → Open Project... from the default Code window main menu. This displays the Select Project to Open dialog box.
4. Locate the required project settings file from the Examples directory, that is \dhrystone\dhrystone.prj, and click Open to load the project details into RealView Debugger.

The project directory becomes the current working directory. This enables you to find new source files, or to set up build options more quickly.

**Default binding**

The dhrystone example project specifies ARM-ADS as the toolchain associated with the target application. Because you are connected to an ARM target, RealView Debugger attempts to bind the project using default binding. If the project binds successfully, the default Code window title bar is updated with the project name to show that it is bound to the current connection:

RVDEBUG(dhrystone) = @ARM7TDMI_0:ARM-A-RR [Unattached]

Because RealView Debugger has the image details, the File Editor pane contains a hyperlink ready to load the associated image.
If you open a second project that specifies the same toolchain, for example \dataabort\dataabort.prj, RealView Debugger gives you the option to unbind the first project and bind the second. In this example, click Cancel. This opens the dataabort example project but maintains dhrystone as the active project and the current binding remains unchanged.

See Project binding on page 11-90 for more details on binding operations and how to control project binding.

**Upgrading the project**

RealView Debugger determines the location of your ARM build tools automatically (see Managing your build tools on page 11-18 for details). If you open a project and a later version of the toolchain exists, RealView Debugger prompts you to upgrade the project.

In this example, it is not necessary to upgrade the example project(s) at this stage. You can do this later if required. See Upgrading the project toolchain on page 11-25 for more details.

### 11.4.2 Generated makefiles

Project makefiles are generated when you do any of the following:

- create a new user-defined project and then save and close the Project Properties window
- edit and close the Build-Tool Properties window
- edit and close the Project Properties window
- edit and resave a user-defined project
- add source files to a user-defined project from the default Code window.

You can also force makefiles to be regenerated by deleting them from the project base directory and then rebuilding the image. Where appropriate, RealView Debugger generates a makefile for each build target configuration in your project (see Managing build target configurations on page 11-72).

RealView Debugger displays the details of the makefile generation process in the Build tab of the Output pane.
Viewing the makefile

You can view the makefile that is generated using RealView Debugger in several ways:

- Select File → Open... from the default Code window main menu. This displays the Select File to Open dialog box, where you can locate the makefile and open it in the File Editor pane for viewing.
- Open Windows Explorer and navigate to the project base directory. Drag the makefile and drop it into the File Editor pane where it opens for viewing.
- Start up a text editor of your choice and then open the makefile for viewing.

Do not make any changes to the generated makefiles as these will be overridden when the files are next generated. It is recommended that you always use the Project Properties window to set up your preferences. However, you can also edit the template file, for example gen_arx.mk for building with ARM compilation tools.

11.4.3 Project properties

When you first create a project, RealView Debugger creates the project settings file and completes the entries to provide the project definition. RealView Debugger dynamically updates this file as you amend the project properties.

The name of the project settings file is defined by the project name and this is stored in the project base directory specified when you created the project. In general, the project settings file remains in this location. However, RealView Debugger uses relative pathnames so you can change the base directory for the project if required.

By default, the executable built by the project also uses the project name as the image name, for example dhrystone.axf. Although your project filename can be different from your image name, it might be advantageous to use the same name for your project settings file as your image name when developing your applications.

To view project properties for the active project dhrystone.prj, select Project → Project Properties... from the default Code window main menu. This displays the Project Properties window.

See Managing project properties on page 11-46 for details on entries in the Project Properties window.

11.4.4 Adding files to a user-defined project

You can add files to the active project from the:

- Project menu
- Project Properties window.
If the Project Properties window is visible, selecting an option from the Project menu might display an error dialog box instructing you to use the window to make the update. You must also wait for any build process to complete before trying to add files to the active project.

Adding files using the Project menu

With a project open, you can add source files to the project and so update the project properties automatically:

1. Choose the menu option according to the location of the file you want to add:
   - Select Project → Add This File to Project to add the file currently displayed in the File Editor pane to the list of sources specified for the project.
   - Select Project → Add Files to Project... to display the Select File(s) to Add dialog box where one or more files can be located and added to the project.

The relative pathname of the file is added to the Sources group in the top-level COMPILE or ASSEMBLE group. If this is the first update to the Sources group, an asterisk is appended to the beginning of the group name.

The Project Properties window is not displayed, but you can view the change by opening the window as described in Project properties on page 11-23.

If your project has more than one COMPILE or ASSEMBLE group enabled, a list selection box is displayed that lists the groups for the type of file being added:
   - for files with extension .c, .cpp, .cc, or .cxx, the list shows enabled COMPILE groups
   - for files with extension .s, .src, or .asm, the list shows enabled ASSEMBLE groups.

Figure 11-3 shows the COMPILE groups for the example dhrystone project.
2. Choose the required group, for example \texttt{COMPILE=arm} and, then click \textbf{OK} to close the dialog box.

The \textbf{Build} tab shows the makefile being regenerated.

3. When the makefile has been regenerated, select \textbf{Tools $\rightarrow$ Build...}, to rebuild the image.

\begin{enumerate}
\item Note
\end{enumerate}

If you select files of different types, for example, $*.c$ and $*.s$ files, then RealView Debugger adds the files to the appropriate group, even if that group is disabled. In this example, the $*.c$ files are added to the \texttt{COMPILE=arm} group, and the $*.s$ files are added to the \texttt{ASSEMBLE=arm} group.

\section*{Adding files using the Project Properties window}

For examples of adding files using the Project Properties window, see:
\begin{itemize}
\item \textit{Adding source files to a project} on page 11-53
\item \textit{Adding object files} on page 11-62
\item \textit{Adding library files} on page 11-64.
\end{itemize}

Also, see \textit{Specifying paths to include files} on page 11-64 if you have project-specific include files in a different directory from your main source files.

\subsection*{11.4.5 Upgrading the project toolchain}

If you have upgraded your build tools, you can upgrade your projects to use the new toolchain. For example, if you currently have ADS v1.2 and you upgrade to RVCT v2.0, you can upgrade your existing ADS projects to use the RVCT toolchain.

If you open a project and RealView Debugger detects that a later version of the toolchain exists, it prompts you to upgrade the project. In this case, RealView Debugger displays the Upgrade Project ToolChain dialog box, shown in Figure 11-4.
Do one of the following:

- Click **Yes** if you want to upgrade your project. This displays the Upgrade Project selection box. See *Using the Upgrade Project selection box* on page 11-27 for details.

- Click **No** if you do not want to upgrade your project now. In this case, RealView Debugger remembers the state of the **Don’t ask me again** check box. If you want to upgrade the project later, then leave this check box unchecked.
  
  RealView Debugger opens the project.

- Click **Cancel** if you do not want to upgrade your project.
  
  RealView Debugger opens the project.

When a project is successfully upgraded, the dialog box is not displayed when you next open that project, unless you upgrade your build tools again. A backup copy of the old project settings file is created automatically.

--- **Note** ---

If you select **Don’t ask me again** for a project, and you want to upgrade that project later, you can use the Project Control dialog box to upgrade the project manually. See *Using the Project Control dialog box* on page 11-81 for details.
Using the Upgrade Project selection box

The Upgrade Project selection box, shown in Figure 11-5, enables you to upgrade the toolchain for a user-defined project. The name of the project you are about to upgrade appears at the top of the selection box and the display list shows available upgrades.

To upgrade the project:
1. Select the conversion you require from the list.
2. Click OK.

A backup copy of the old project settings file is created automatically.

11.4.6 Closing a user-defined project

As you change a user-defined project, add new files, or update dependencies, the project properties are updated and saved automatically. It is not necessary to close your project to update the associated project settings file.

Select Project → Close Project... from the default Code window main menu to close an open project. If the associated Project Properties window is displayed, this closes automatically when the project closes.

If you have more than one project open, RealView Debugger displays a selection box where you can specify which project to close, for example dhrystone. If you have two projects open, the second project, for example dataabort, becomes the active project when the first project closes.

When you close a project, the default Code window title bar is updated to show the new active project, where applicable. The current working directory remains as defined by the last file access. You can change this by resetting it or when you open another project from a different location.
Project binding

When you close a bound project, it unbinds automatically. If you had two projects open, the second project, for example dataabort, does not bind by default because this only happens when a project opens to a connection. You can, however, rebind the project if required.

Note

If you are connected when you close the project, any close commands you have specified are executed. If you are not connected when the project closes, these commands are not run. See Command_Open_Close group on page B-6 for more details.

See Project binding on page 11-90 for more details on binding operations and how to rebind a project.

Working with images

If you close a user-defined project and the image is not loaded, RealView Debugger removes all image details. This clears the hyperlink in the File Editor pane.

If you close a user-defined project where the image is loaded, RealView Debugger prompts you to unload the image. Click Yes to unload the image and remove all image details.

If you do not unload the image, RealView Debugger searches for a saved auto-project to provide project properties for the image. If no file exists for the image, RealView Debugger creates an in-memory auto-project to use in this session. This binds to the connection by default.
11.5 Creating a new user-defined project

The procedure for creating each type of user-defined project has a common sequence of steps (see Common steps for creating a user-defined project). The specific steps for each user-defined project are described in the individual sections.

If you have modified and saved settings for an auto-project, then you can merge these settings into a new user-defined project during the creation procedure.

Note

If you do want to merge auto-project settings, make sure that you read the explanation in Merging auto-project settings into a project on page 11-36 before you start to create the new project.

This section describes:

- Before you start
- Common steps for creating a user-defined project
- Steps for creating a Standard or Library project on page 11-31
- Steps for creating a Custom project on page 11-32
- Steps for creating a Container project on page 11-33
- Steps for copying an existing user-defined project on page 11-35
- Customizing and building your project on page 11-35
- Merging auto-project settings into a project on page 11-36.

11.5.1 Before you start

Before you follow the procedures described in this section:

1. Start RealView Debugger.
2. Connect to the ARM7TDMI core using ARMulator.
3. Make sure that you have no projects open. You can create new projects with open projects but this affects binding behavior.
4. Set up your build tools as described in Managing your build tools on page 11-18.

11.5.2 Common steps for creating a user-defined project

To create a new user-defined project:

1. Select Project → New Project... from the default Code window main menu to display the Create New Project dialog box.
2. Complete the entries in the Create New Project dialog box:

**Project Name**

Enter a name for your project in this data field.
Check that any name entered here is not already in use for a project.
Where a project with the same name already exists, RealView Debugger gives you the option to replace it.

**Project Base**

Specify the project base directory to use as the location of the project files.
This data field might be preloaded with your RealView Debugger installation directory name as defined by your environment variable. This can be overridden.
Click the folder icon to view the directory chooser, and select *<Select Dir...>* to specify a directory that is not in the recently used directory list.
If the specified directory does not exist, RealView Debugger gives you the option to create it.

**Select Type of Project**

Select the type of project you want to create. For more details about the project types, see *Types of user-defined project* on page 11-4.

3. Click **OK**.

**Caution**

If you confirm that an existing project is to be overwritten, there is no undo and the contents of the first project settings file are lost.

4. RealView Debugger displays a dialog box to create the type of project you selected.

Follow the steps described in the appropriate section to complete the entries:

- *Steps for creating a Standard or Library project* on page 11-31
- *Steps for creating a Custom project* on page 11-32
- *Steps for creating a Container project* on page 11-33
- *Steps for copying an existing user-defined project* on page 11-35.

5. Click **OK** to confirm the project details and close the dialog box for your project type.

6. Make any changes to the project properties to customize the new project. See *Customizing and building your project* on page 11-35.
11.5.3 Steps for creating a Standard or Library project

The steps described in this section assume that you have performed the first three steps described in Common steps for creating a user-defined project on page 11-29.

To create a Standard or Library project:

1. Complete the entries on the Create Standard Project or the Create Library Project dialog box:

   **Project Name**
   This shows the name specified for the new project.

   **Toolchain**
   Choose the toolchain to use for this project from the drop-down list. RealView Debugger uses this to bind the project to all available debug targets that have the same processor type.

   **Sources (C/C++/Assembly) to build from**
   Click the directory icon to locate the sources used to build the executable image or library file. The source files are added to the source list box, and they are all selected.
   Alternatively, enter a filename in the text box, then click the Add button. The file is added to the source file list, and is selected.
   You can perform other operations on the source file list:
   - Click Del to delete all selected files from the source file list.
   - Click Rep to replace the selected file in the source file list with the file specified in the text box.
   - Click AllOn or AllOff to respectively select or deselect all files in the source file list.

   **Note**
   You do not have to add your source files here. You can create the project, and then add the source files later. See Adding files to a user-defined project on page 11-23 for details.

   **Executable**
   RealView Debugger completes this field based on the project name, for example std_proj_1.axf or lib_proj_1.lib. Change this entry if required.
   Alternatively, click on the directory icon to locate the required executable or library file.

   **Description**
   Enter a text description for the new project, saved in the PROJECT group.
2. Click OK to confirm the project details and close the Create Standard Project or the Create Library Project dialog box.

3. Make any changes to the project properties to customize the new project. See Customizing and building your project on page 11-35.

11.5.4 Steps for creating a Custom project

The steps described in this section assume that you have performed the first three steps described in Common steps for creating a user-defined project on page 11-29.

To create a Custom project:

1. Complete the entries on the Create Custom Project dialog box:

   - **Project Name**
     This shows the name specified for the new project.

   - **Toolchain**
     Select the toolchain to use for this project from the drop-down list.
     RealView Debugger uses this to bind the project to all available debug targets that have the same processor type.

   - **Select type of Custom Project**
     Specifies how the image is built for the project and determines what additional information you must provide.

     - **Make a makefile (your own makefile)**
       Uses the default make command together with your makefile to build the image.

     - **Run Command (your own tool/builder)**
       Uses your own build tool command to build the image. You must enter the command in the Command field.
       If you want to use the command expansion controls, $a, $e, and $f, then you must specify the information in the corresponding field.

       ——— Note ———
       To build your Custom project successfully, you must remove the $e control character, and use your own arguments as required. See Using your own make command on page B-79.

     - **No-Build (echo Arguments as message only)**
       If you do not want to build an image for this project.
Command

Note

The Command field must be filled even when the project specifies a no-build model, for example use a dummy entry such as dummy. Do not enter any other command here, for example to run a batch file.

This data field contains the default make command used for the new project, for example, `make -f $f $a $e`.

You can also use the project path expansion control `$p` in the command, for example, `make -f $p$f`. The path expansion control uses the path you specified in the Project Base data field, see Common steps for creating a user-defined project on page 11-29 for details.

See Appendix B Project Properties Reference for details on how to amend this default makefile command.

Use the fields below to populate the makefile command line or set up the command manually.

File Arg

This data field contains the name of the makefile, that is `$f`.

Arguments

This data field contains the arguments to the command, that is `$a`.

Executable

This data field contains the executable file to build, that is `$e`.

Description

This data field contains a text description for the new project, saved in the PROJECT group.

2. Click OK to confirm the project details and close the Create Custom Project dialog box.

3. Make any changes to the project properties to customize the new project. See Customizing and building your project on page 11-35.

11.5.5 Steps for creating a Container project

The steps described in this section assume that you have performed the first three steps described in Common steps for creating a user-defined project on page 11-29.
To create a Container project:

1. Complete the entries on the Create Container Project dialog box:

   **Project Name**
   This shows the name specified for the new project.

   **Sub-Projects (order defines build order)**
   Click the directory icon to locate the project files to use for the Container project. The project settings files are added to the project list box, and they are all selected.
   Alternatively, enter a project filename in the text box, then click the **Add** button. The file is added to the project file list, and is selected.
   You can perform other operations on the project file list:
   - Click **Del** to delete all selected files from the project file list.
   - Click **Rep** to replace the selected file in the project file list with the file specified in the text box.
   - Click **AllOn** or **AllOff** to respectively select or deselect all files in the project file list.

   **Note**
   You do not have to add your subprojects here. You can create the project, and then add the subprojects later using the Project Properties window.

   **Description**
   Enter a description for the project, saved in the PROJECT group.

2. Click **OK** to confirm the project details and to close the Create Custom Project dialog box.

3. Make any changes to the project properties to customize the new project. See Customizing and building your project on page 11-35.

The nature of Container projects means that if you have a Container project open, you are working on multiple projects at the same time. You can perform additional operations on the individual projects in a Container project. See Working with Container projects on page 11-105, for more details.

   **Note**
   Container projects can be nested but not recursive, that is a Container project can include other Container projects but must not include itself. However, the nested Container project has no makefile and so any build fails.
11.5.6 Steps for copying an existing user-defined project

The steps described in this section assume that you have performed the first three steps described in *Common steps for creating a user-defined project* on page 11-29.

To create a copy of an existing user-defined project:

1. Locate the project settings file to copy on the Select Project to Copy from dialog box.
   
   By default, RealView Debugger begins the search in the destination location because this is now the current working directory.

2. Click **Open** to copy the project details to the new project.

3. If the project you are using to create the new project has associated subdirectories, you are prompted to copy the directory tree structure. In general, it is recommended that you copy the existing tree structure for the new project:
   - Click **Yes** to copy the directory tree to the new project. RealView Debugger replicates the entire contents starting at the location of the specified .prj file.
   - Click **No** if you do not want to copy the tree.

4. If you are copying a Container project, RealView Debugger displays a list of any subprojects that could not be opened. Make a note of these projects. You can add these projects using the Project Properties window.
   
   Click **OK** to close the dialog box.

5. Make any changes to the project properties to customize the new project. See *Customizing and building your project*.

11.5.7 Customizing and building your project

When you close the dialog box for your project type, RealView Debugger:

1. Creates the .prj file in the specified location.

2. Opens the project into the debugger.

3. Binds the project to the current connection. If a project is already bound to the connection, RealView Debugger gives you the option to unbind the first project and bind the second. Although it is not necessary, bind the new project to complete the creation procedure.

4. Updates the default Code window title bar to show the active project.

5. Displays the Project Properties window for you to customize your project.
The entries shown in this window depend on the type of project you create.

To complete your new project:

1. Make any changes you require in the Project Properties window, see *Managing project properties* on page 11-46 for details.

2. Select **File** → **Save and Close** to close the Project Properties window and generate the makefile(s) for the new project.

3. Select **Tools** → **Build...** from the default Code window main menu to build the active project.

### 11.5.8 Merging auto-project settings into a project

If you have an image and you have modified, and saved, an auto-project file associated with that image, then you can merge those settings with the settings for a new project.

__________ **Note** __________

Merging only gives you the option to incorporate the SETTINGS group from your auto-project. If you change settings in any other group, for example the Command_Open_Close commands in the PROJECT group, these are lost when the auto-project merges.

__________

To merge an auto-project you must have:

- an image, for example \test_examples\test_image.axf
- a saved auto-project for the image, for example \test_examples\test_image.axf.apr.

By definition, an auto-project is a no-build project and, therefore, uses no build model. You can merge an auto-project into the following types of project:

**Standard** Create a new Standard project to merge the settings and use a build model.

**Custom** Create a new Custom project to merge the settings and use your own makefile or build tools. This project uses no build model.

See *Using a build model* on page 11-41 for more details on build models.

This section describes:

- **Merging options** on page 11-37
- **Common steps for merging auto-project settings** on page 11-38
- **Steps for merging settings into a Standard project** on page 11-38
Steps for merging settings into a Custom project on page 11-39
Customizing and building your project on page 11-39.

Merging options

If you have made changes to load-related values in the SETTINGS group of your auto-project, you can choose whether to merge these settings into the properties of the new project. During the creation process, a list selection box is displayed that enables you to specify the merge, shown in Figure 11-6.

![Figure 11-6 Merging options selection box](image)

The merging options are:

**Merge in settings and delete Auto-project file**
Merges the SETTINGS group from the auto-project into the new project and then deletes the auto-project.

**Merge in settings**
Merges the SETTINGS group from the auto-project into the new project but keeps the auto-project.

**Delete Auto-project file**
Deletes the auto-project without merging the SETTINGS group.

**Leave Auto-project alone**
Keeps the auto-project without merging the SETTINGS group.

---

**Note**
Choose this if you have not saved any settings for your auto-project. This is the same as clicking **Cancel**.
Common steps for merging auto-project settings

Create the user-defined project you require:

1. Follow the first three steps in Common steps for creating a user-defined project on page 11-29 with these entries:
   a. Enter the name of your project in the Project Name data field.
   b. Enter the directory containing your auto-project file in the Project Base data field, for example \test_examples.
   c. Select the type of project to create, either:
      • Standard
      • Custom.

2. RealView Debugger displays a dialog box to create the type of project you selected.
   Follow the steps described in the appropriate section to complete the entries:
   • Steps for merging settings into a Standard project
   • Steps for merging settings into a Custom project on page 11-39.

3. Make any changes to the project properties to customize the new project. See Customizing and building your project on page 11-39.

Steps for merging settings into a Standard project

To merge your auto-project settings:

1. Complete the entries in the Create Standard Project dialog box as described in the first step in Steps for creating a Standard or Library project on page 11-31.
   You must specify the full pathname of the image associated with the auto-project in the Executable data field, for example \test_examples\test_image.axf.

2. Click OK to display the Project Properties window.

3. If RealView Debugger locates the auto-project for the specified image, it displays a list selection box where you can choose the merging option (see Merging options on page 11-37 for details):
   a. Select the required option from the list.
   b. Click OK to update the new project settings file as requested. If selected, the auto-project file is also deleted.

4. Make any changes to the project properties to customize the new project. See Customizing and building your project on page 11-39.
Steps for merging settings into a Custom project

To merge your auto-project settings:

1. Complete the entries in the Create Custom Project dialog box as described in the first step in Steps for creating a Custom project on page 11-32.
   You must specify the full pathname of the image associated with the auto-project in the Executable data field, for example \test_examples\test_image.axf.
2. Click OK.
3. If RealView Debugger locates the auto-project for the specified image, it displays a list selection box where you can choose the merging option (see Merging options on page 11-37 for details):
   a. Select the required option from the list.
   b. Click OK to update the new project settings file as requested. If selected, the auto-project file is also deleted.
4. Make any changes to the project properties to customize the new project. See Customizing and building your project.

Customizing and building your project

When you close the merging options selection box, RealView Debugger:

1. Creates the .prj file in the specified location.
2. Opens the project into the debugger.
3. Binds the project to the current connection, if possible.
4. Updates the default Code window title bar to show the active project.
5. Displays the Project Properties window for you to customize your Standard project. If you merge into a Custom project, select Project → Project Properties... to display this window.

To complete your new project:

1. Make any changes you require in the Project Properties window, see Managing project properties on page 11-46 for details.
2. Select File → Save and Close to close the Project Properties window and generate the makefile(s) for the new project, where applicable.
3. Select Tools → Build... from the default Code window main menu to build the active project.
--- Note ---

When you build a new image, the image is placed in the directory defined by the active build model for the project, for example `test_examples\Debug`. Your original image remains in the project base directory, for example `\test_examples`. 
11.6 Building your application

After you have created your user-defined project, you have to build an executable file or image from the project components. When creating a Standard project, all these stages are completed automatically from RealView Debugger presets. You can, however, manually create a project when you want to have more control over the build process.

Also, each time you make a change to settings using the Project Properties window, the makefiles are updated. The next time you build the image, RealView Debugger asks for confirmation before rebuilding.

--- Note ---
You do not have to have a project open to build an image. RealView Debugger includes a Build dialog box where you can specify what to build.

The tutorial in RealView Debugger v1.6 Essentials Guide introduces building an image, but the build process, and how to customize it, are covered in more detail in this section and in Configuring project properties on page 11-52.

This section describes the build options in more detail. It includes:
- Using a build model
- Resetting line numbers on page 11-42
- Building files and images on page 11-42
- Using the Build dialog box on page 11-44
- Finding build errors on page 11-45.

11.6.1 Using a build model

For a Standard or Library project, the build model specifies how the image is built, that is the build target configuration. The Project Properties window enables you to view, and change, the rules governing the build model as defined in the makefile generated by RealView Debugger.

Defining rules for a build model

You define the rules for a build model using groups in the Project Properties window:

**CONFIGURATION**

 Define build target configurations in this group.

**COMPILE** Specify compiler options in this group.
ASSEMBLE Specify assembler options in this group.

CUSTOM Specify custom build options in this group.

BUILD For a Standard project, specify linker options in this group.

BUILD_LIB For a Library project, specify library options in this group.

In addition, you can use your own makefiles for the build, or use your own build tool automatically.

For more details on project properties see:

- Managing project properties on page 11-46
- Configuring project properties on page 11-52
- Managing build target configurations on page 11-58.

11.6.2 Resetting line numbers

When building starts, line numbers might be reset. This depends on your Editing Controls settings.

Make sure that you have a source file displayed in the File Editor pane. Select Edit → Editing Controls to display the Editing Controls menu.

By default, the options Show Line Numbers and Use Original Line Numbers are not selected. However, if you select Use Original Line Numbers, this enables debugging to continue while editing and makes error tracking easier.

With these settings checked, building automatically resets the original line numbers of the source code for all files in the default Code window. If other files are showing in other Code windows that are part of the build, their line numbers are not reset.

11.6.3 Building files and images

For a project, you can build an executable image or a single object file. If you have a Library project, you can build the library file or an individual object file.

Note

If you do not have a project open, you can still use the build options on the Tools menu.
Building an object file

To build an object file:

1. Open a source file so that it is displayed in the File Editor pane. This can be a C, C++, or assembly language file.

2. Select Tools → Build This File to display the Build dialog box shown in Figure 11-7.

   ![Figure 11-7 Building an object file](image)

   Entries might be preloaded into fields in the dialog box when it first opens. In this example, the file is associated with a user-defined project.

3. Enter the build details (see Using the Build dialog box on page 11-44).

4. Click Build to build the object file.

   Alternatively, click Cancel to close the dialog box without starting the build.

5. After the object file is built, you can include it into an executable file for later loading to your debug target processor.

Building an executable image or library file

To build an image:

1. Make sure that there is no open project and no files in the File Editor pane.

2. Select Tools → Build... to display the Build dialog box shown in Figure 11-8 on page 11-44.
Entries might be preloaded into fields in the dialog box when it first opens.

3. Enter the build details (see Using the Build dialog box).

4. Click either Build or Build+Load (see Using the Build dialog box).
   Alternatively, click Cancel to close the dialog box without starting the build.

11.6.4 Using the Build dialog box

The Build dialog box enables you to build images, object files or library files from the default Code window. RealView Debugger preloads fields in the dialog box if it can. The controls and options available depend on:

- the type of target you are building
- which option you select from the Tools menu
- what files are displayed in the File Editor pane
- the project environment, that is what projects are open.

The Build dialog box controls are:

**Local Filelist (for Targets)**

Lists the currently loaded images, if any. This enables you to make a quick selection. Available only if you select Tools → Build... to build an executable image.

**Working Directory**

Use the radio buttons to specify where to set the working directory.
### Target
Enter the target file, for example `sample_arm.obj`, if not preloaded from the project. You can also click on the drop-down arrow to display a list of previously used build targets.

### Makefile
Enter a makefile name, for example `sample_arm_Debug.mk`. Click on the directory icon to locate the required pathname or select from a list of previously used makefiles.

### Directory
If the **Specified** radio button is selected, use this data field to enter the name of the working directory. Click on the directory icon to locate the required directory or select from a list of previously used files.

### Build Command
As you enter the build details, the command is constructed and displayed in this data field. You can amend the command line before submission. You can also click on the drop-down arrow to display a list of previously used command lines.

### Build
Builds the object file or executable image. Depending on the target, this also completes the compile and assemble stages.

### Build+Load
Builds the executable image. If the image is built successfully, RealView Debugger loads the image to the debug target processor. Available only if you select **Tools → Build...** to build an executable image.

### Cancel
Closes the dialog box without starting the build.

### 11.6.5 Finding build errors
The build error reporting system in RealView Debugger:

1. Lists the errors in the **Build** tab of the Output pane.

2. If the source file containing the error is not displayed in the File Editor pane, RealView Debugger opens it automatically ready for you to correct the error.

3. Positions the flashing text cursor in the source file tab of the File Editor pane, at or near the line containing the error. A blue arrow in the left margin of the File Editor pane also indicates this position.

When you have corrected the first error, and RealView Debugger has found more than one error, locate the next error in the source file using one of the following methods:

- select **Tools → Next Line/Error** to move through the error list
- double-click on the line number shown in the Output pane **Build** tab.
11.7 Managing project properties

To examine project settings for the active project, select **Project → Project Properties...** from the default Code window main menu. This displays the Project Properties window, as described in **Project Properties window** on page 11-47.

You can also display the Project Properties window from the Process Control pane if you are connected to a debug target. See **Managing projects in the Process Control pane** on page 11-84 for more details.

This section describes the contents of the Project Properties window. For detailed information on changing entries, see **Configuring project properties** on page 11-52 and **Managing build target configurations** on page 11-72.

This section includes:

- Viewing project properties
- **Project Properties window** on page 11-47
- **Groups available for each type of project** on page 11-49
- **Disabling a group** on page 11-50
- **Viewing the Configuration Summary** on page 11-50.

11.7.1 Viewing project properties

Use the Project Properties window to view and change the project settings file. This contains the settings values, or rules, that describe a project. In some cases, these values are preset and cannot be changed. In other cases, you can amend the preloaded settings, or define your own values to customize your project.

If an entry has been changed from the default setting, an asterisk (*) is appended to the group name to show that the contents have been updated. You can choose to view only those settings that you have changed from the default. To do this, select **View → Show Default Entries** to disable the menu option. When the menu option is enabled, all settings are available for viewing.

--- Note ---

The settings shown in the Project Properties window vary depending on the type of project. For example, if your project is a Custom project using a custom makefile, the window contains the PROJECT, SETTINGS, and MAKEFILE groups.
11.7.2 Project Properties window

The Project Properties window enables you to customize your project in the same way that you configure other settings files, such as build tool properties and workspace settings. This window contains many of the same controls and menus that are used to configure settings described in other parts of the RealView Debugger documentation. For general instructions on how to use these common controls and menus, see the Changing Settings topic in the RealView Debugger online help. The rest of this section describes the controls that are specific to setting up your projects.

Select **Project → Project Properties...** from the default Code window main menu to display the Project Properties window, shown in Figure 11-9.

![Figure 11-9 Project Properties window](image)

Figure 11-9 shows the properties for a Standard project, for example `dhrystone.prj`. Other types of project have a different set of properties.

The left pane of the Project Properties window, the List of Entries pane, shows project settings as a hierarchical tree with node controls. The right pane, the Settings Values pane, displays the contents of any group that you select in the left pane.

The first setting in the List of Entries pane, the left pane, of the Project Properties window is the name of the project settings file, for example `...\dhrystone.prj`. When selected, the Description field shows the full pathname of this project settings file.

Other settings below the project settings file specify groups of settings. Each group defines that part of the build model covered by the contents. For example, in a Standard project the `CONFIGURATION` group defines the build target configurations and the `COMPILE` group defines the compilation stage.
Where appropriate, a project might contain multiple groups, for example an interworking project that uses both ARM and Thumb code. Use unique names to identify multiple groups for this type of project.

If a group is disabled, the entry in the left pane is grayed out, shown in Figure 11-9 on page 11-47. You can delete disabled groups but this is not necessary. Any group that is disabled is ignored for the project, even if entries in the group are set (see Disabling a group on page 11-50 for details). If you delete a disabled group, it might be harder to change the project later.

Where permitted, groups can be deleted, renamed or copied. Making a copy of an existing group creates a group that can then be edited in the usual way. If the group you are copying is marked by an asterisk, the copy is also marked in the same way.

You can also make a new group so that you can configure the build model to your specification and choose appropriate names for the groups in the project. Making a new group creates a new container for the default settings. See Configuring project properties on page 11-52 for examples of how to work with groups.
11.7.3 Groups available for each type of project

Table 11-1 shows the groups that are available for the different project types. See Appendix B Project Properties Reference for detailed information about the contents of these groups.

<table>
<thead>
<tr>
<th>Project type</th>
<th>Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>PROJECT SETTINGS</td>
</tr>
<tr>
<td></td>
<td>CONFIGURATION</td>
</tr>
<tr>
<td></td>
<td>COMPILE</td>
</tr>
<tr>
<td></td>
<td>ASSEMBLE</td>
</tr>
<tr>
<td></td>
<td>BUILD</td>
</tr>
<tr>
<td>Library</td>
<td>PROJECT SETTINGS</td>
</tr>
<tr>
<td></td>
<td>CONFIGURATION</td>
</tr>
<tr>
<td></td>
<td>COMPILE</td>
</tr>
<tr>
<td></td>
<td>ASSEMBLE</td>
</tr>
<tr>
<td></td>
<td>CUSTOM</td>
</tr>
<tr>
<td></td>
<td>BUILD_LIB</td>
</tr>
<tr>
<td>Custom and auto-project</td>
<td>PROJECT</td>
</tr>
<tr>
<td></td>
<td>SETTINGS</td>
</tr>
<tr>
<td></td>
<td>MAKEFILE</td>
</tr>
<tr>
<td>Container</td>
<td>PROJECT</td>
</tr>
</tbody>
</table>
11.7.4 Disabling a group

To disable a group in the Project Properties window:

1. Right-click on the group to be disabled, for example *COMPILE=arm*, and select Explore from the context menu.
2. Right-click on the Disable setting in the Settings Values pane and select True from the options list.
3. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.
4. Select Tools → Build... to rebuild the application.

If you disable a group, the entry in the left pane is grayed out but it does not appear as an entry in the right pane of the Project Properties window, shown in Figure 11-9 on page 11-47. You can view and change entries in a disabled group in the usual way, see Configuring project properties on page 11-52 for details.

To enable the group again:

1. Right-click on the disabled group, for example *COMPILE=arm*, and select Explore from the context menu.
2. Right-click on the Disable setting in the Settings Values pane and select False from the options list.
3. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.
4. Select Tools → Build... to rebuild the application.

11.7.5 Viewing the Configuration Summary

Use the Configuration Summary to see a read-only display of the switches that RealView Debugger is passing to the compilation tools (compiler, assembler, and linker) for each build target configuration in a project. This information is displayed in a tabbed window. When you make a change that affects the switches of a tool, the list of switches in the Configuration Summary window for that tool is updated immediately.

To view the list of switches:

1. In the Project Properties window, select the group that relates to the tool of interest:
   - select a COMPILE group to view the switches for the compiler
   - select an ASSEMBLE group to view the switches for the assembler
• select the BUILD group to view the switches for the linker of a Standard project
• select the BUILD_LIB group to view the switches for the linker of a Library project.

2. Select View → Configuration Summary to see the Configuration Summary window below the Project Properties window. Figure 11-10 shows an example configuration summary for the ARM C compiler.

3. To view the switches for a specific build target configuration, select the tab for that configuration.

4. To close the Configuration Summary window, click the X button, shown in the top-right corner of the window in Figure 11-10.

Note
If you close the Project Properties window when the Configuration Summary window is open, then both windows close. When you next open the Project Properties window then the Configuration Summary window is also displayed.
11.8 Configuring project properties

You can change your project configuration by editing the project settings file using the RealView Debugger Project Properties window (see Project Properties window on page 11-47).

This section describes:
- Customizing your project
- Adding source files to a project on page 11-53
- Removing source files from a project on page 11-54
- Compiling a specific source file on page 11-55
- Excluding source files from a build on page 11-55
- Changing the location of object files on page 11-56
- Changing build tools for a specific user-defined project on page 11-57
- Managing build target configurations on page 11-58
- Using MS-DOS names under Windows on page 11-58
- Using MS-DOS applications on page 11-59
- Customizing the build on page 11-59
- Adding object files on page 11-62
- Adding library files on page 11-64
- Specifying paths to include files on page 11-64
- Adding prelink and postlink commands on page 11-65
- Specifying breakpoints on page 11-66
- Specifying linker options and scatter loading on page 11-67
- Interworking ARM and Thumb on page 11-68.

11.8.1 Customizing your project

The rest of this section contains examples of making changes to the example user-defined project dhrystone.prj installed in the Examples directory in your root installation. You might want to make a backup of the project base directory before following the examples so that the default files and settings can be restored. The changes described in Specifying breakpoints on page 11-66 can also be applied to an auto-project.

These examples assume that you are not connected to a debug target. This means that RealView Debugger does not try to set default binding. See Project binding on page 11-90 for more details.
Caution

The changes described here conflict with the default build target configurations as specified in the example project. These examples are included only to show how to amend project and build-tool properties.

Before you start

To start working with the example dhrystone project:
1. Make a copy of the project base directory if necessary.
2. Start up RealView Debugger.
3. Select Project → Open Project... from the default Code window main menu.
4. Locate the project settings file dhrystone.prj and click Open.

Note

If you are using the example dhrystone project, you can still follow the steps for adding sources, but you must then remove the duplicate source files before continuing with the other tasks. See Removing source files from a project on page 11-54.

11.8.2 Adding source files to a project

To add source files to an existing project:

1. Select Project → Project Properties... to display the Project Properties window.
2. Right-click on the required group, in the List of Entries pane, that defines the build stage that you want to modify, for example «COMPILE=arm», and select Expand whole Tree from the context menu.

   You can also expand, and collapse, groups in the List of Entries pane by clicking on the plus sign, or the minus sign, at each node in the tree. If you double-click on the group name, this expands the group and displays the contents in the Settings Values pane.

3. Select the *Sources group in the List of Entries pane.
4. Right-click on the default Files setting, at the top of the list, and select Edit as Filename from the context menu.

   This displays the Enter New Filename dialog box where you can locate source files to add to the project. Select the source file you want to add and click Save.
Alternatively, right-click on the default Files setting, at the top of the list, and select Manage List... from the context menu. Use the Settings: List Manager dialog box to modify the source list as required. You can also use this dialog box to add more files to, or to remove files from, the group. Click OK when you have finished adding files using the Settings: List Manager dialog box.

5. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.

6. Select Tools → Build... to rebuild the application.

You can also use this method to add files to other groups, for example other COMPILE groups, ASSEMBLE groups or the BUILD group.

Before adding files to the active project in this way, you must wait for any build process to complete and the results to be visible in the Output pane.

Note
You do not have to add .h files to a project, because these are referenced from the main source files for your project. See Specifying paths to include files on page 11-64.

### 11.8.3 Removing source files from a project

To remove source files from a project:

1. Select Project → Project Properties... to display the Project Properties window.

2. Right-click on the required group, in the List of Entries pane, that defines the build stage that you want to modify, for example «COMPILE=arm», and select Expand whole Tree from the context menu.

3. Select the «Sources» group in the List of Entries pane.

4. Right-click on the «Files» setting for the source file that is to be removed, and select Delete from the context menu.

   The setting value for the source file is deleted.

5. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.

6. Select Tools → Build... to rebuild the application.
11.8.4 Compiling a specific source file

To compile a specific source file that is part of an existing project:

1. Select Project → Project Properties... to display the Project Properties window.
2. Right-click on the required group, in the List of Entries pane, that defines the build stage that you want to modify, for example *COMPILE=arm, and select Expand whole Tree from the context menu.
3. Select the *Sources group in the List of Entries pane.
4. Right-click on one of the source files and select Compile File... from the context menu. This recompiles the file and displays any debugger messages in the Output pane. It is not necessary to close the Project Properties window to view these messages.
5. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.
6. Select Tools → Build... to rebuild the application.

11.8.5 Excluding source files from a build

To exclude a source file from a build:

1. Select Project → Project Properties... to display the Project Properties window.
2. Right-click on the required group, in the List of Entries pane, that defines the build stage that you want to modify, for example *COMPILE=arm, and select Expand whole Tree from the context menu.
3. Select the *Sources group in the List of Entries pane.
4. Right-click on the *Files setting for the source file that is to be excluded, and select Exclude this file from Build from the context menu.
   The setting is grayed out and an exclamation mark (!) is added to the start of the filename.
5. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.
6. Select Tools → Build... to rebuild the application.
Re-including a source file into a build

To re-include a source file into a build:

1. Select **Project → Project Properties...** to display the Project Properties window.
2. Right-click on the required group, in the List of Entries pane, that defines the build stage that you want to modify, for example *COMPILE*=arm, and select **Expand whole Tree** from the context menu.
3. Select the *Sources* group in the List of Entries pane.
4. Right-click on the *Files* setting for the source file that is currently excluded, and select **Re-Include this in Build** from the context menu.
   The setting color is restored and the exclamation mark is removed from the start of the filename.
5. Select **File → Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.
6. Select **Tools → Build...** to rebuild the application.

11.8.6 Changing the location of object files

To change the location of object files built for the project:

1. Select **Project → Project Properties...** to display the Project Properties window.
2. Select the *COMPILE*=arm group in the List of Entries pane.
3. Right-click on the **Obj_location** setting in the Settings Values pane, and select **sub_dir** from the options list.
   Making this change means that object files are located in a directory called `\objects` inside the project base directory.
4. Right-click on the **Obj_sub** setting in the Settings Values pane, and select **Edit Value** from the context menu.
5. Type `test_objects` and press Enter to set this new destination location.
6. Select **File → Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.
7. Select **Tools → Build...** to rebuild the application.
8. Use Windows Explorer to view the newly-built object files in the specified location in the project base directory.
This might be useful if you want to have different output from two projects using the same source files and same base directory. You can specify the location of the object files so that a potentially dangerous mix-up does not happen, for example when building the two applications in sequence. You can also control object files using different CONFIGURATION groups as described in Managing build target configurations on page 11-58.

--- Note ---

Be careful here if multiple projects share the same directory or use the same source paths for output files.

### 11.8.7 Changing build tools for a specific user-defined project

When you install RealView Debugger for the first time, it determines the location of your build tools to use for all user-defined projects that you create. However, you can override this setting or specify a different tool for selected projects.

To change the compiler for a project:

1. Select **Project → Project Properties...** to display the Project Properties window.
2. Select the *COMPILE=arm* group in the List of Entries pane.
3. Right-click on the Tool_path setting in the Settings Values pane, and select **Edit as Filename** from the context menu.
4. Use the Enter New Filename dialog box to locate the compiler for this project.
5. Click **Save** to confirm your choice and to enter the pathname.
6. Select **File → Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.
7. Select **Tools → Build...** to rebuild the application.

To restore the changed setting:

1. Select **Project → Project Properties...** to display the Project Properties window.
2. Select the *COMPILE=arm* group in the List of Entries pane.
3. Right-click on the Tool_path setting in the Settings Values pane, and select **Reset to Empty** from the context menu.
4. Select **File → Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.
5. Select **Tools → Build...** to rebuild the application.

### 11.8.8 Managing build target configurations

For any project, the build target configurations define how the files are processed, and enable you to build the same application in different ways. The default build target configurations available depend on the toolchain defined for the project. For example, for ARM-based projects, RealView Debugger provides the following default build target configurations:

- **Debug**
- **DebugRel**
- **Release**

See *Build target configurations* on page 11-7 for a description of these configurations.

You can also create your own build target configurations, and assign different customized settings to each configuration. For example, you can assign different compilation controls for the **ARM C compiler** to each configuration. For detailed instructions on managing your build target configurations, see *Managing build target configurations* on page 11-72.

### 11.8.9 Using MS-DOS names under Windows

You can amend your project properties to accommodate tools that cannot handle:

- long filenames, that is, names greater than eight characters
- spaces in filenames.

To change this behavior:

1. Select **Project → Build-Tool Properties** to display the Build-Tool Properties window.
2. Select the **PROC=** group corresponding to your build tools in the List of Entries pane. For example, **PROC=ARM_ADS** if you have ADS.
3. Right-click on the **Dos_names** setting in the Settings Values pane, and select **always** from the options list.
4. Select **File → Save and Close** to regenerate the makefile(s) for the project, and close the Build-Tool Properties window.
5. Click **OK** at the prompt to regenerate the project because of the change to the toolchain.
6. Select **Tools → Build...** to rebuild the application.
11.8.10 Using MS-DOS applications

When running older 16-bit tools, you might see a range of symptoms during different stages of a build. For example, slow echoing of keystrokes, disk access errors, or rogue windoldapp tasks left in the system. You can set RealView Debugger to support these older tools that use extended memory, and run your old tools in a DOS dialog box.

**Note**

Do not set this option for new 32-bit applications because this severely impacts performance.

To change this behavior:

1. Select **Project** → **Build-Tool Properties** to display the Build-Tool Properties window.
2. Select the *PROC* group corresponding to your build tools in the List of Entries pane. For example, *PROC=ARM_ADS* if you have ADS.
3. Right-click on the `Dos_app` setting in the Settings Values pane, and select **True** from the context menu.
4. Select **File** → **Save and Close** to regenerate the makefile(s) for the project, and close the Build-Tool Properties window.
5. Click **OK** at the prompt to regenerate the project because of the change to the toolchain.
6. Select **Tools** → **Build...** to rebuild the application.

11.8.11 Customizing the build

To output a message during the build:

1. Select **Project** → **Project Properties...** to display the Project Properties window.
2. Right-click on the *COMPILE=arm* group in the List of Entries pane, and select **Make New...** from the context menu to display the Group Type/Name selector dialog box.
3. Highlight **CUSTOM** in the Group Type display list.
4. Type **MY_GROUP** in the Group Name field.
5. Click **OK** to close the Group Type/Name selector dialog box.
6. Select the `CUSTOM=MY_GROUP` group in the List of Entries pane, and select **Explore** from the context menu.

7. Click on the **Message** setting, in the Settings Values pane, and select **Edit Value** from the context menu.

8. Type `Writing version to version file` and then press Enter to confirm the value. This string is output every time you build or rebuild the application.

9. Right-click on the **Files** setting, in the Settings Values pane, and select **Edit Value** from the context menu.

10. Type `version.txt` and then press Enter to confirm the value.

11. Right-click on the **Depends_on** setting, in the Settings Values pane, and select **Edit Value** from the context menu.

12. Type `$(MAKENAME)` and then press Enter to confirm the value. Ensure that you do not include any spaces in the entry. This entry means that output file is dependent on the makefile so you rebuild version.txt each time the makefile is updated. Usually, dependent files are inputs to a build but this example illustrates the method.

   Do not make the executable file, dhrystone.axf, the dependent because that does not work. Instead, you must use postlink commands as shown in Adding prelink and postlink commands on page 11-65.

13. Right-click on the **Command** setting, in the Settings Values pane, and select **Edit Value** from the context menu.

14. Type `+echo 'version = 1.00 >$@'` and then press Enter to confirm the value. This command writes the message to the text file defined previously. You might use a command to run another program or to output the date.

   Preceding the command with a plus sign means that the command is a built-in operating system shell command. Do not put a plus sign in the command when running normal applications. If you do not want the command to be shown while it is running, then put an at (@) sign in front of the command string.

   The Project Properties window is shown in Figure 11-11 on page 11-61.
15. Select **File → Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.

16. Select **Tools → Build...** to rebuild the application.

**Running the build**

You can control the build using the `use_as` settings value, shown in Figure 11-11. Right-click on the setting to see the available options:

**link_dependent**

*RealView Debugger ensures that it is up-to-date when building the application. This is the default.*

**link_input**  
*Creates an object file or library which is then linked to the application built by the project.*

**named_target**  
*Makes the file a named target. This means that other files must be dependent on it for it to be used. Use this option to create a header or source file, that is a `.c` or `.asm` file, that is then compiled or assembled in a `COMPILE` or `ASSEMBLE` group. The make knows to build the header or source file before building the application with it. You can also layer custom projects in this way with the output of one project used as the input of another.*

**post_link**  
*Specifies that the file is built only after linking.*
Note

You can force a CUSTOM rule to run every time that you build the application by referring to a file that does not exist. For example, in the steps in Customizing the build on page 11-59, you could have specified that the file was called version but still written to version.txt. This causes the CUSTOM rule to be executed each time you build the application because make decides that the file version does not exist and so tries to build it.

To undo this change and restore the build model:

1. Select Project → Project Properties... to display the Project Properties window.
2. Right-click on the $CUSTOM=MY_GROUP group in the List of Entries pane, and select Delete from the context menu.
3. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.
4. Select Tools → Build... to rebuild the application.

11.8.12 Adding object files

The Objects setting defines the location of object files, as specified in the COMPILE group.

To add object files to the build:

1. Select Project → Project Properties... to display the Project Properties window.
2. Select the $BUILD group in the List of Entries pane.
3. To add a single object:
   a. Right-click on the Objects setting in the Settings Values pane, and select Make New from the context menu.
   b. Use in-place editing to enter the pathname of the required object file files.
   c. Confirm the entry to see the new object file added to the list. For example, C:\Add_proj_2\Objects\thumbtest.o, as shown in Figure 11-12 on page 11-63.

Note

You can also select Edit as Filename to locate the required object file.
4. To add several object files:
   a. Right-click on the Objects setting, in the Settings Values pane, and select Manage List... from the context menu. This displays the Settings: List Manager dialog box.
   b. Click Add for each new object file to be added.

   **Note**
   You can also use the Settings: List Manager dialog box to sort and reorder the list, for example where object link order is important.

5. Click OK to confirm the entries. The Settings: List Manager dialog box closes, and the object files list in the Project Properties window is updated.

6. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.

7. Select Tools → Build... to rebuild the application.
11.8.13 Adding library files

If you have library files that are not in the normal library search path defined by the `Lib_path` settings value and the `ARMLIB` environment variable, then you must include the location of those library files. The `Libraries` setting defines the location of library files.

To add library files to the build:

1. Select **Project → Project Properties...** to display the Project Properties window.
2. Select the `BUILD` group in the List of Entries pane.
3. To add a single library:
   a. Right-click on the `Libraries` setting in the Settings Values pane, and select **Make New** from the context menu.
   b. Use in-place editing to enter the pathname of your library files.
   c. Confirm the entry to see the new library file added to the list. For example, `C:\Add_proj_2\Libraries\lib_proj_1.a`, as shown in Figure 11-12 on page 11-63.

   **Note**
   You can also select **Edit as Filename** to locate your library files.

4. To add several library files:
   a. Right-click on the appropriate `Libraries` setting, in the Settings Values pane, and select **Manage List...** from the context menu. This displays the Settings: List Manager dialog box.
   b. Click **Add** for each new library file to be added, enter the path and filename for the library file, then click **Add** to add it to the list.
5. Click **OK** to confirm the entries. The Settings: List Manager dialog box closes, and the library paths list in the Project Properties window is updated.
6. Select **File → Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.
7. Select **Tools → Build...** to rebuild the application.

11.8.14 Specifying paths to include files

For C and C++ sources, if you have project-specific header files and these are in a different directory from the rest of your sources, you must specify the path to include the files. For assembler sources, you can specify source file search paths.
To specify paths to include files that are in different directories to your main source files:

1. Select Project → Project Properties... to display the Project Properties window.
2. Right-click on the appropriate *COMPILE or *ASSEMBLE group in the List of Entries pane, and select Expand whole Tree from the context menu.
4. Right-click on Include in the List of Entries pane, and select Edit as Directory Name from the context menu. The Enter New Directory dialog box is displayed.
5. Select the required directory, then click Select.
6. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.
7. Select Tools → Build... to rebuild the application.

### 11.8.15 Adding prelink and postlink commands

Prelink and postlink commands are run only when the linker is run. Prelink commands can be used to copy libraries or objects from other developers so that you do not have to build them yourself. Postlink commands might be used to convert the executable file into another format, for example to load to Flash or for loading by a specific operating system.

To add prelink and postlink commands to the build:

1. Select Project → Project Properties... to display the Project Properties window.
2. Right-click on the *BUILD group in the List of Entries pane, and select Expand whole Tree from the context menu.
3. Select the Pre_Post_Link group in the List of Entries pane.
4. Right-click on the Pre_link setting in the Settings Values pane, and select Edit Value from the context menu.
5. Type `@+echo This is before linkage` and then press Enter to confirm the value. This string is output every time you build or rebuild, the application, but the command is not displayed when it is executed.
6. Right-click on the Post_link setting in the Settings Values pane, and select Edit Value from the context menu.
7. Type `+copy $(PROGRAM) $\temp.axf` and then press Enter to confirm the value.
Type this exactly as shown.

This postlink command copies the output from the build, that is dhrystone.axf, to another location. In this case, you are sending the executable file to a temporary location, but it might be a predefined central location for images to be debugged. The copy command is displayed in the Build tab when executed.

8. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.

9. Select Tools → Build... to rebuild the application.

11.8.16 Specifying breakpoints

Projects enable you to control an application that you are debugging. You can include image load commands and set breakpoints that are stored as part of the project. The SETTINGS group includes two groups for breakpoints:

Auto_Set_Breaks

Set as soon as a symbol is matched. If there is no symbol specified, these are set on any load. These breakpoints appear in the Break/Tracepoints pane in the usual way.

Named_Breaks

Breakpoints that you want to set often. Enables you to set breakpoints that are specific to the application, to an RTOS, or to a library.

Note

The steps for adding Auto_Set_Breaks and Named_Breaks are identical, except that for Auto_Set_Breaks you can choose to have RealView Debugger prompt you before setting the breakpoint. The steps in this section describe setting Named_Breaks.

To add named breakpoints:

1. Select Project → Project Properties... to display the Project Properties window.

2. Select the SETTINGS group in the List of Entries pane.

3. Right-click on the Named_Breaks group in the Settings Values pane, and select Explore from the context menu.

4. Right-click on the Default group in the Settings Values pane, and select Make Copy... from the context menu.

5. Enter a new name for the group, for example My_breakpoints, and click Create.
6. Right-click on the My_breakpoints group in the Settings Values pane, and select Explore from the context menu.

7. Right-click on the Cmd setting, in the Settings Values pane, and select Edit Value from the context menu.

8. Type bi \DHRY_1\#149:5 and then press Enter to confirm the value. This identifies a software breakpoint on the chosen instruction.

9. Right-click on the Description setting, in the Settings Values pane, and select Edit Value from the context menu. The text entered here appears in the list selection box to identify the named breakpoint.

10. Type My test breakpoint and then press Enter to confirm the value. This text identifies the named breakpoint.

11. Create a new group, for example My_test_breakpoints, and set up a second named breakpoint if required.

12. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.

13. Select Tools → Build... to rebuild the application.

14. Connect to your target and load the newly-built image dhrystone.axf.

15. Click on the Src tab to view the source file dhry_1.c.

16. Select Debug → Simple Breakpoints → Named... to display the list selection box. This box lists the named breakpoints you previously set up. Select the breakpoints you want to set, then click OK. These breakpoints are also available when you next open the project.

When you select breakpoints from the Named Breakpoints list, they appear in the Break/Tracepoints pane in the usual way.

**11.8.17 Specifying linker options and scatter loading**

The default output from the ARM linker is a non-relocatable image where the code starts at 0x8000 and the data section is placed immediately after the code. You can specify exactly where the code and data sections are located by using linker options or a scatter load descriptor file.

To set linker options:

1. Select Project → Project Properties... to display the Project Properties window.
2. Right-click on the BUILD group in the List of Entries pane, and select **Expand whole Tree** from the context menu.

3. Select the Link_Advanced group in the List of Entries pane.

4. Right-click on Scatter_file in the Settings Values pane, and select **Edit as Filename** from the context menu to locate a previously created scatter file for the project image (see your build tools documentation for details on creating scatter files). Alternatively, specify other linker options as required.

5. Select **File → Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.

### 11.8.18 Interworking ARM and Thumb

You can mix C and C++ code for ARM and Thumb, provided that the code conforms to the requirements of the ARM and Thumb Procedure Call Standards. See *ARM-Thumb Procedure Call Standard (ATPCS) Specification* for details.

The ARM linker detects when ARM and Thumb code is mixed and generates small code segments, called *veneers*, that control the change in state between ARM and Thumb.

If you compile a module for interworking, it generates slightly larger code for Thumb, around 2%, and marginally larger code for ARM. Use the linker option `-info` to find the amount of space taken up by the veneers. Disabled by default, this can be set in the BUILD group for your interworking project.

---

**Note**

ARM code compiled for interworking cannot be used on ARM processors that are not Thumb-capable, for example StrongARM, because these processors do not implement the BX (Branch Exchange) instruction.

---

This section contains examples of making changes to the example project `interworking.prj` located in the `\Examples` directory in your root installation. You might want to make a backup of the project base directory before following the examples so that the default files and settings can be restored.

The examples describe:

- **Setting compiler options for interworking** on page 11-69
- **Displaying code sizes** on page 11-71.
To see an example of interworking ARM and Thumb code:

1. Select **Project → Open Project...** from the default Code window main menu to open the example project `interworking.prj` located in the `\Examples` directory of your root installation.
2. Open the source file `hello_goodbye.c` so that it is displayed in the File Editor pane.
3. Add a new line just before the call to the Thumb routine, for example:
   
   ```c
   24 fprintf(stdout,"Now in Thumb routine\n");
   25 thumbtest();
   ```
4. Select **Tools → Build...** to rebuild the application. This displays a list selection box where you can confirm the source file that has been changed.
5. Click **OK** to confirm the save and rebuild the application.
6. Connect to a Thumb-capable debug target, for example to the ARM7TDMI core using ARMulator.
7. Load the image `interworking.axf`, and run the application. The **StdIO** tab, in the Output pane, displays the results.

——— **Note** ————

The interworking example project contains an unused deck of cards to show how RealView Debugger displays structures and arrays, see `hello_goodbye.c`. This is not specific to interworking.

**Setting compiler options for interworking**

To set compiler options for ADS:

1. Select **Project → Project Properties...** to display the Project Properties window. Expand the entries, shown in Figure 11-13 on page 11-70.
2. Confirm the compiler options for the interworking example project.

The *Interworking settings value is set to enabled for the *COMPILE=arm group. This sets the -apcs /interwork compiler option. This means that the ARM compilers can compile modules containing routines called by other routines compiled for Thumb state.

Similarly, expand the *COMPILE=thumb group to see the APCS setting enabled for routines compiled for ARM state.

3. The interworking example project defines two enabled COMPILE groups, shown in Figure 11-13:
   - COMPILE=arm specifies the ARM C compiler, that is armcc, to compile ANSI C source into 32-bit ARM code
   - COMPILE=thumb specifies the Thumb C compiler, that is tcpp, to compile ANSI C source into 16-bit Thumb code.

The COMPILE=arm_cpp settings value is grayed out showing that it is disabled for this project. This specifies the ARM C++ compiler, that is arnccpp, to compile ANSI C++ or EC++ source into 32-bit ARM code.

4. If you want to use software stack checking, this can be set in this settings values page. Set to the default for the compiler, you can force checking on or off.

   Right-click on the Stack_checking setting in the Settings Values pane, and select enabled.
Ensure that all ARM and Thumb modules are compiled to the same standard if they are to be interworked. Failure to do this results in a warning from the compiler, for example:

Error: L6242E: Cannot link object thumbtest.o as its attributes are incompatible with the image attributes.
   ... stack-checked clashes with not-stack-checked.

5. Select **File → Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.
6. Select **Tools → Build...** to rebuild the application.

**Displaying code sizes**

To create a file containing code sizes for your interworking project:

1. Select **Project → Project Properties...** to display the Project Properties window.
2. Right-click on the **BUILD** group at the bottom of the List of Entries pane, and select **Expand whole Tree** from the context menu. This displays the contents in the Settings Values pane.
3. Select the **Listings** group in the List of Entries pane.
4. Right-click on the **Listing_file** setting in the Settings Values pane, and select **Edit as Filename** from the context menu. Enter the path of a text file to accept the linker output, for example ...\interworking\listing.txt.
5. Select the **Messages** group in the List of Entries pane.
6. Right-click on the **Sizes** setting, in the Settings Values pane, and select **both** from the options list. This displays details and totals size information.
7. Right-click on the **Veneers** setting, in the Settings Values pane, and select **enabled** from the options list.
8. Select **File → Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.
9. Select **Tools → Build...** to rebuild the application.
10. Open the linker messages file in the File Editor pane and view the contents.
11.9 Managing build target configurations

This section describes how to manage build target configurations for user-defined projects:

- *About build target configurations*
- *Creating a build target configuration on page 11-74*
- *Deleting a build target configuration on page 11-74*
- *Changing the active build target configuration on page 11-75*
- *Changing the order of build target configurations on page 11-75*
- *Assigning a specific setting to a build target configuration on page 11-76*
- *Removing a setting from a build target configuration on page 11-77*
- *Assigning multiple settings to a specific build target configuration on page 11-79.*

Note

The rest of this section contains examples of making changes to the example user-defined project dhrystone.prj installed in the \Examples directory of your root installation. You might want to make a backup of the project base directory before following the examples so that the default files and settings can be restored.

11.9.1 About build target configurations

A user-defined project defines at least one build target configuration, for example a
Debug build or a Release build. For ARM-based projects, RealView Debugger defines
three build target configurations:

- Debug
- DebugRel
- Release.

See *Build target configurations* on page 11-7 for a description of these configurations.

You can define a specific build order for the build target configurations in a project.
Build target configurations can share files in the same project, while using their own
build settings. The Project Properties window in RealView Debugger enables you to
define and set up such relationships.

Each build target configuration has a corresponding directory where the built files are
placed. The directories have the same name as the build target configuration, and are
subdirectories of your top-level project base directory.
Viewing the build target configurations

For a given project, each COMPILE, ASSEMBLE, and CUSTOM group contains special child groups corresponding to the build target configurations defined in the CONFIGURATION group. In addition, each parent group contains base settings that apply to one or all of the build target configurations listed, shown in Figure 11-14. You can customize your project so that settings apply to all the build target configuration groups or only to specific groups within the parent group.

Figure 11-14 Base settings and build target configurations for a COMPILE group

Figure 11-14 shows the *COMPILE=arm group for the example dhrystone project. This group contains settings that define the build model for the ARM C compiler, that is:

- Base settings used in one or more build target configurations. If you specify a base setting, this is applied across all the build target configurations in the group.
- You can define specific settings for individual configurations within the group.
- Build target configuration groups used to hold configuration-specific settings, for example the *debugRel group.

A special icon identifies these groups because they are internal groups generated by RealView Debugger. If you create additional build target configurations, they appear in this window. See Creating a build target configuration on page 11-74 for an example of how to do this.
Note

Build target configurations are also available in the BUILD group of a Standard project, and the BUILD_LIB group of a Library project.

Active build target configuration

If you have more than one build target configuration, you must specify the configuration that RealView Debugger uses when building your application. This is known as the active build target configuration. See Changing the active build target configuration on page 11-75 for details on how to change the active build target configuration.

11.9.2 Creating a build target configuration

The CONFIGURATION group, shown in Figure 11-14 on page 11-73, defines the build target configurations for the project.

To create a new build target configuration:

1. Select Project → Project Properties... to display the Project Properties window.
2. Select the *CONFIGURATION group in the List of Entries pane.
3. Right-click on the Config setting in the Settings Values pane, and select Make New... from the context menu.
4. Type the name of your new configuration, for example myBuilds, then press Enter. A new *Config setting is created in the CONFIGURATION group. This adds a new group, called myBuilds, to every build target configuration group specified for the project. This applies to disabled groups.
5. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.
   RealView Debugger creates a makefile (for example, projectname_myBuilds.mk) and a subdirectory for the new build target configuration in the project base directory.
6. Select Tools → Build... to rebuild the application.

11.9.3 Deleting a build target configuration

To delete a build target configuration:

1. Select Project → Project Properties... to display the Project Properties window.
2. Select the *CONFIGURATION group in the List of Entries pane.
3. Right-click on the *Config setting to be removed and select **Delete**.
   This deletes the specified group from the CONFIGURATION group. The group is also removed from every build target configuration group for the project.

   **Note**
   The makefile and build directory for the deleted configuration are not deleted, where they exist.

4. Select **File** → **Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.
5. Select **Tools** → **Build**... to rebuild the application.

### 11.9.4 Changing the active build target configuration

By default, the active build target configuration is **Debug**. You can specify which build target configuration is the active configuration:

1. Select **Project** → **Project Properties**... to display the Project Properties window.
2. Select the *CONFIGURATION group in the List of Entries pane.
3. Right-click on the Active_config setting in the Settings Values pane, and select the required configuration from the options list, for example **DebugRel**.
4. Select **File** → **Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.
5. Select **Tools** → **Build**... to rebuild the application.

### 11.9.5 Changing the order of build target configurations

If you work with build target configurations in a particular sequence, you can arrange the settings so they appear in that sequence in the Project Properties window. For example, you might start by building a Debug configuration, then a DebugRel configuration, and finally a Release version.

To change the order of the build target configurations:

1. Select **Project** → **Project Properties**... to display the Project Properties window.
2. Select the *CONFIGURATION group in the List of Entries pane.
3. Right-click on any *Config setting in the Settings Values pane, and select **Manage List...** from the context menu.

The Settings: List Manager dialog box is displayed.

4. Select a build target configuration that you want to move, so that the associated tick box is checked.

5. Click **Move Up** or **Move Down** to move the select configuration as required.

Repeat these steps for each configuration you want to move.

6. Click **OK** to confirm the order and close the Settings: List Manager dialog box.

7. Select **File → Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.

8. Select **Tools → Build...** to rebuild the application.

### 11.9.6 Assigning a specific setting to a build target configuration

Each build target configuration can be assigned a different value for a particular setting. For example, you might want to have a different value for the **Speed_vs_space** setting in each configuration.

---

**Note**

For projects that support build target configurations, the context menu for settings in the Settings Values pane includes an option called **Move/Copy to Configuration...** This option is valid only for the settings in the **COMPILE**, **ASSEMBLE**, **CUSTOM**, **BUILD** and **BUILD_LIB** groups. You must not use this option on settings in the **PROJECT** and **SETTINGS** groups.

---

To assign a specific setting value to a build target configuration:

1. Select **Project → Project Properties...** to display the Project Properties window.

2. Select the group containing the setting to be assigned. For example:
   a. Select the **COMPILE=arm** group in the List of Entries pane.
   b. Right-click on the **Optimization** group in the Settings Values pane, and select **Explore** from the context menu.

3. Modify the setting to be assigned. For example:
   a. Right-click on the **Speed_vs_space** setting in the Settings Values pane.
   b. Select **space** from the context menu.
If you want to copy the setting to more than one build target configuration, you must modify the setting first. Only settings that have been changed from the default can be copied to other configurations.

4. Right-click on the Speed_vs_space setting, and select **Move/Copy to Configuration...** from the context menu.
   A list selection box is displayed that shows the actions available. There is a move and copy action for each of your build target configurations.

5. Select the required action, for example **Copy to Debug**.
   - If you have not modified the setting from the original default value, then the setting is always moved to the build target configuration, even if you choose to copy the setting.
   - If you have modified the setting, then the action you selected is performed.
   In this example, a copy of the setting is assigned to the Debug build target configuration.

6. Select **File → Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.

7. Select **Tools → Build...** to rebuild the application.

**Note**
This change only applies to this *Optimization* group. It does not replicate throughout other groups in the project.

### 11.9.7 Removing a setting from a build target configuration

When you assign a setting to a build target configuration, and an instance of the setting also exists in the base settings, the assigned setting value overrides the base settings value. If you now want the base settings value to be used for the configuration, you can remove the assigned setting.

For example, if you have completed the procedure described in **Assigning a specific setting to a build target configuration** on page 11-76, you might now want to remove the Speed_vs_space setting.

To remove a setting from a build target configuration:

1. Select **Project → Project Properties...** to display the Project Properties window.
2. Select the group containing the setting to be removed. For example:
   a. Select the *COMPILE=arm group in the List of Entries pane.
   b. Right-click on the *Optimization group in the Settings Values pane, and select Explore from the context menu.

3. Right-click on the Speed_vs_space setting in the Debug build target configuration to display the context menu.

4. Select an option to reset the setting:
   - Select Move/Copy to Configuration... if you have not modified this instance of the setting from the default value. This enables you to move the setting into the base settings (see Moving a setting into the base settings).
   - Select either Reset to Default or Reset to Empty if you have modified this instance of the setting from the default value. Only one of these options is available.

   The result of selecting a Reset to ... option depends on other settings in this group:
   - If there are other instances of the chosen setting in this settings group, this instance of the setting is deleted from the build target configuration.
   - If this is the only instance of the chosen setting in this settings group, the value of the setting is changed according to the menu option selected. This instance of the setting remains in the build target configuration. That is, there is no instance in the base settings.

5. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.

6. Select Tools → Build... to rebuild the application.

Moving a setting into the base settings

If there is no instance of a setting in the base settings for a settings group, and you want that setting to apply to more than one build target configuration, then you must move the setting into the base settings. For example you might want to move a setting from the DebugRel configuration so that it applies to all build target configurations in this settings group.

To move a setting from a build target configuration to the base settings:

1. Select Project → Project Properties... to display the Project Properties window.
2. Select the group containing the setting to be moved. For example:
   a. Select the `*COMPILE=arm` group in the List of Entries pane.
   b. Right-click on the `*Optimization` group in the Settings Values pane, and select Explore from the context menu.

3. Right-click on the setting in the required build target configuration. For example:
   a. Right-click on the `*Debug_optimize partial setting in the `DebugRel` configuration.
   b. Select Move/Copy to Configuration... from the context menu.

4. Select Move to <Base> from the selection box, and click OK.

5. Select File → Save and Close to regenerate the makefile(s) for the project, and close the Project Properties window.

6. Select Tools → Build... to rebuild the application.

11.9.8 Assigning multiple settings to a specific build target configuration

To assign multiple settings to a build target configuration:

1. Select Project → Project Properties... to display the Project Properties window.

2. Select the group containing the setting to be assigned. For example:
   a. Select the `*COMPILE=arm` group in the List of Entries pane.
   b. Right-click on the `*Optimization` group in the Settings Values pane, and select Explore from the context menu.

3. Right-click on the build target configuration group to which the settings are to be assigned, for example `Debug`, and select the action you require from the context menu:
   - **Insert Item(s) into this Configuration...**
     Inserts a new instance of a setting into the specified configuration. The value of the new instance is the default value of the setting. The original instance of the setting remains unchanged.
   - **Move Item(s) into this Configuration...**
     Moves a current instance of a setting into the specified configuration. This maintains the current value of the setting.
   - **Copy Item(s) into this Configuration...**
     Copies a current instance of a setting into the specified configuration. This maintains the current value of the setting.
4. Select the setting(s) to be assigned to the chosen build target configuration so that the associated tick boxes are checked:
   • if you choose to insert settings, the default settings are listed
   • if you choose to move or copy settings, then only the modified settings are listed, if any.

   __________ Note __________

   If there is more than one instance of a setting, then each instance of the setting is listed, and they are in the same order that they appear in the Settings Values pane.

5. Click **OK** to confirm your choices and assign the setting(s).

6. Select **File** → **Save and Close** to regenerate the makefile(s) for the project, and close the Project Properties window.

7. Select **Tools** → **Build...** to rebuild the application.

For full information on all the settings described in these examples see Appendix B *Project Properties Reference*. 
11.10 Using the Project Control dialog box

You can use the Project and Tools menus to perform operations on your open projects. By default, these menus give you access to the active project. See Using the Project and Tools menus on page 11-12 for details.

Where you are working with multiple projects, you can use the Project Control dialog box to act on other projects or to change the project environment.

The Project Control dialog box enables you to:
• control project binding
• display the Project Properties window for a project
• perform build, rebuild, or clean actions on a project
• change the active project
• upgrade a project to use a new toolchain.

This section describes:
• Viewing the Project Control dialog box
• Working with the Project Control dialog box on page 11-82.

11.10.1 Viewing the Project Control dialog box

Select Project → Project Control... from the default Code window main menu to display the Project Control dialog box shown in Figure 11-15.

![Figure 11-15 Project Control dialog box](image)

The Project Control dialog box displays the open project list. When RealView Debugger opens a user-defined project, or an auto-project, it is added to the top of the open project list.
The open project list shows the order in which the projects were opened. If you are not connected when you open projects, the most recent project, the default active project, is at the top of the list. This is selected by default. If you are connected, the projects are in the same order but the last project to open is not necessarily the active project, see Working with multiple projects for details.

Working with multiple projects

If you are connected and you open multiple projects, the open project list shows which project is bound to the current connection. In Figure 11-15 on page 11-81, there are three projects in the open project list. The last project to open is shown at the top of the list and is selected by default. The dataabort project is bound to the ARMulator connection, ARM7TDMI_0:ARM-A-RR, using default binding. In this case, dataabort is the active project because it was the last project to bind successfully to the current connection.

The Project Control dialog box enables you to carry out operations on all open projects. In some cases, you can complete actions on several projects, for example building, but other actions can only be carried out on a single project, for example using the Edit button to view the project settings file.

If you select multiple projects from the list and then click an action control, the specified action is carried out on the selected projects in the order they are listed and so, in some cases, the last action completed successfully overwrites all previous actions.

Note
If you are licensed to work in multiprocessor debugging mode, the open project list shows how projects are bound to all active connections.

11.10.2 Working with the Project Control dialog box

To perform an action on a project:

1. Select Project → Project Control... from the default Code window main menu to display the Project Control dialog box.

2. Click on a project in the list so that the associated check box is checked. This selects a project for action.

3. Use the controls in the dialog box to operate on the chosen project:
   Re-Bind Enables you to bind a project to a specific connection. This is most useful when you are working with multiple projects and multiple connections. See Forcing binding on page 11-97 for details.
Changes made here are reflected in the **Process** tab in the Process Control pane, if this is visible.

**Un-Bind**
Enables you to unbind a project from the current connection. This is most useful when you are working with multiple projects and multiple connections. See *Forcing binding* on page 11-97 for details.

Changes made here are reflected in the **Process** tab of the Process Control pane, if this is visible.

**Edit**
Displays the Project Properties window for the chosen project ready for editing. This is independent of the active project.

**Build**
Enables you to build one or more applications. If one build fails, RealView Debugger asks for confirmation before continuing with the other builds listed. Projects are built in the order shown in the display list.

**Clean**
Enables you to clean one or more projects. Projects are cleansed in the order shown in the display list.

**Rebuild**
Enables you to clean and rebuild one or more applications. If one rebuild fails, RealView Debugger asks for confirmation before continuing with the other rebuilds. Applications are rebuilt in the order shown in the display list.

**Active**
Makes the chosen project the active project if you have more than one project open. The active project is reflected in the title bar in the default Code window that changes to show the new active project. The project binding remains unchanged. See *Active projects* on page 11-103 for more details.

**Upgrade**
Displays the Upgrade Project dialog box, where you can upgrade the project to use a new toolchain. If you try to upgrade a project where no upgrades exist, RealView Debugger displays a message box to say that no upgrades are available. See *Upgrading the project toolchain* on page 11-25 for more details.
11.11 Managing projects in the Process Control pane

If you are connected to a debug target, use the Process Control pane to see details about the current process and target processor. If you open a project that binds to the current connection, the Process Control pane also enables you to view your project properties and perform operations on the project files.

This section describes:
- Displaying the Process Control pane
- Working with user-defined projects on page 11-85
- Working with auto-projects on page 11-85
- Project context menus on page 11-86.

11.11.1 Displaying the Process Control pane

To display the Process Control pane, shown in Figure 11-16, select View → Pane Views → Process Control Pane from the Code window main menu.

In Figure 11-16 the:
- Process Control pane is floating and so the title bar reflects the calling Code window
- Process tab shows details about the target processor
- image defined by the active project, dhrystone.axf, is not loaded
- user-defined project dhrystone is bound to the connection.

Note
The Settings entry for user-defined projects is always set to <Saved>. 
11.11.2 Working with user-defined projects

When you create or open a Standard or Custom user-defined project, the project name is shown in the Project entry in the Process Control pane. The example in Figure 11-17 shows the contents for the example dhrystone project.

![Figure 11-17 User-defined projects in the Process Control pane](image)

The groups shown in the Process Control pane correspond to groups in the Project Properties window, see Figure 11-14 on page 11-73. All these groups are shown, that is groups that are disabled in the Project Properties window are visible here. These groups do not affect the build model.

11.11.3 Working with auto-projects

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

--- Note ---

If you have created a project, it is recommended that you open this first to load and debug the associated image. This enables you to build your application and to change project settings.

---

Figure 11-18 on page 11-86 shows a Process Control pane where the image dhrystone.axf has been loaded to the debug target without opening the project first.
In this example, the Project <Auto> entry shows that RealView Debugger is using an auto-project. An in-memory auto-project is identified by the Settings <Not Saved> entry.

When you save the settings for an auto-project, they are saved in a .apr file, for example, dhrystone.axf.apr. An auto-project that has a saved settings file is identified by a Settings <Saved> entry.

11.11.4 Project context menus

You can perform project-level operations using context menus available from the Process Control pane. Right-click on the Project, Settings, or Sources entries, shown in Figure 11-18, to display the context menus. For details on these operations see:

- **Build operations**
- **Project and Settings operations** on page 11-87
- **Source file operations** on page 11-88
- **Working on a source file** on page 11-89.

_____ Note  ___________

If you have several projects open, any unbound projects are usually inaccessible from the Process Control pane by definition. See **Project binding** on page 11-90 for details on how to work on these projects.

---

**Build operations**

These operations are available on all the project context menus, unless it is a no-build project or an auto-project:

**Build**

Builds the application.
Managing Projects

Rebuild All  Rebuilds the application.

Clean       Cleans the project.

Update Dependencies

Updates all dependencies for the project makefile.

Properties Displays a text box showing information about the project. The box
details a subset of the PROJECT group defined at creation.

These operations are also available from the main menu bar. See Using the Project and
Tools menus on page 11-12 for details.

Project and Settings operations

These options are available, in addition to the build operations, when you right-click on
the Project or Settings entry in the Process Control pane:

Close       Closes the project. The action depends on whether the project is a
user-defined project or an auto-project:

• For a user-defined project, if the image is loaded when you close
the project, you are prompted to unload the image. If you choose
not to unload the image, RealView Debugger closes your
user-defined project, and creates an in-memory auto-project for the
image. Otherwise, RealView Debugger unloads the image and
closes the project.

• For an auto-project, RealView Debugger closes the project even if
the image is loaded.

If the Project Properties window is visible, and you have any unsaved
changes, RealView Debugger gives you the option to save the changes
before the project closes.

Closing a project automatically unbinds it from the connection.

Save        Saves the in-memory settings for the auto-project to a file called
image_name.axf.apr, and places it in the same location as the image file.
This option is available only for in-memory auto-projects, that is the
Settings entry indicates <Not Saved> and no auto-project settings file
exists.

Delete Auto-Project File

Deletes the settings file, image_name.axf.apr, for the auto-project. The
Settings status in the Process Control pane changes to <Not Saved>.
Delete the auto-project settings file to force RealView Debugger to create
a new in-memory auto-project when you next load the image.
This option is only available for a saved auto-project.

**Project Properties...**

Displays the Project Properties window that enables you to edit the settings for the project.

**Source file operations**

The Process Control pane displays a list of source files for the current project. Right-click on the Sources entry to specify how RealView Debugger collects this list to populate this pane. The operations available, in addition to the build operations, depend on the type of project:

**Collect from Project**

Where selected, this indicates that the list of source files shown is derived from the project. The sources are listed in the appropriate COMPILE and ASSEMBLE groups.

This is selected by default for a Standard project.

**Collect from Makefile**

Select this option if you want to use a project makefile to define which source files are shown.

This is selected by default for a Custom project.

**Collect from Image**

Select this option if you want to use the image to define which source files are shown. The image must be loaded for the sources to be shown.

This is the only option for an auto-project.

--- **Note** ---

Files might be listed that are not included in the list of files for your project. This is because the file list comes from the image, and so the source paths might be unknown, or unavailable. For example, standard C or C++ library files might be listed.

---

You must load the image before you change the way that source files are collected if you want to collect from the image.
Working on a source file

You can work on a source file directly from the Process Control pane. Right-click on a file, for example \texttt{dhry\_1.c} and then choose the required option from the context menu. The options available depend on how the debugger collects the list of files (see \textit{Source file operations} on page 11-88):

- **Open File**: Opens a new tab in the File Editor pane in the default Code window and displays the chosen file for editing. If the selected file is already in the File Editor pane, then the tab is brought to the top for editing.
  
  This option is available when sources are collected from either the project or makefile.

- **Scope to File**: Scopes to the chosen file.
  
  This option is available only when sources are collected from the image.
  
  You can also scope to a specified file by double-clicking on the filename in the Process Control pane.

- **Properties**: Displays a text box showing details about the source file.
11.12 Project binding

RealView Debugger binds projects to connections by associating the image for a user-defined project or auto-project with an available debug target that has a corresponding processor type. The binding mechanism can also load the project image automatically, and execute RealView Debugger commands, depending on settings in the project settings file.

When you are working with multiple projects, or if you are licensed to work in multiprocessor debugging mode, you can control project binding manually to have full control over your project environment. See Using the Project Control dialog box on page 11-81 for details.

This section describes project binding in more detail:

- Types of binding
- Viewing project binding on page 11-95
- Forcing binding on page 11-97
- Effects of binding on page 11-99
- Effects of unbinding on page 11-100
- Connecting and disconnecting on page 11-100.

11.12.1 Types of binding

RealView Debugger binds a project using either:

- Default binding
- Autobinding on page 11-91.

Default binding

When you first create a Standard, Library, or Custom project, you define the toolchain, for example ARM-ADS, associated with the target application using the Toolchain drop-down list box on the Create New Project dialog box (see, for example, Steps for creating a Standard or Library project on page 11-31). This entry, in the project settings file, determines the default binding for the project.

Note

RealView Debugger populates this setting automatically when it creates an in-memory auto-project for an image that you load directly to a debug target.
If you connect to a debug target and open a user-defined project, RealView Debugger attempts to bind the project by default:

- If the connection does not correspond to the processor family for the project, the project opens but it does not bind. If the project does not bind, the image name is not registered. This means that the debugger cannot autoload the image on a GO or RELOAD command.

  See *SETTINGS group* on page B-9 for more details.

- If the connection corresponds to the processor family and there is no project already bound to the connection, RealView Debugger binds the project by default.

- If the connection corresponds to the processor family and there is a project already bound to the connection by default, RealView Debugger gives you the option to unbind the active project and bind the new project. If you choose not to bind the new project, the project opens but is unbound.

- If the connection corresponds to the processor family and there is a project already autobound to the connection, RealView Debugger displays a warning that it cannot complete the binding. This is because default binding cannot displace an autobound project. The project opens but is unbound. See *Autobinding* for details about autobound projects.

This behavior means that you can always open a user-defined project, and make changes to the project settings file, even if your connection is not a suitable debug target for that project.

**Note**

Default binding also applies in multiprocessor debugging mode. If you are licensed to work in this mode, RealView Debugger examines all active connections to determine whether it can bind a project by default.

*See Effects of binding* on page 11-99 and *Effects of unbinding* on page 11-100 for more details on changing binding.

**Autobinding**

You can specify that a project can only bind to a specific processor or device, by name, using the *Specific_device* entry in the project settings file, shown in Figure 11-19 on page 11-92. You can do this when you first create the project or update this setting later.
This close coupling of project and target processor is called *specific device binding* and forces RealView Debugger to use *autobinding* when the project opens. If autobound, the project only binds to the named device. If this is not set, RealView Debugger uses default binding to determine project binding.

**Note**

You cannot force an autobound project to bind to a device that does not match. You must change the *Specific device* setting first, see *Adding Specific device settings* on page 11-93 for details. It is not necessary to open the project before rebinding. See *Forcing binding* on page 11-97 for details on defining binding behavior.

Autobinding gives a project priority over other projects when it opens to a connection:

- If the connection does not correspond to the named device, the project opens but it does not bind.
- If the connection corresponds to the named device and there is no project already bound to the connection, RealView Debugger autobinds the project.
- If the connection corresponds to the named device and there is a project already bound to the connection, using autobinding or default binding, RealView Debugger unbinds the active project and autobinds the new project. There is no warning.
If the connection partially corresponds to the named device, for example the Specific_device entry is set to ARM, and there is a project already autobound to the connection, for example where the Specific_device entry is set to ARM7, RealView Debugger unbinds the active project and autobinds the new project. There is no warning.

If you open a user-defined project that has no Specific_device entry specified and there is a project already autobound to the connection, RealView Debugger displays a warning that it cannot complete the binding. This is because default binding cannot displace an autobound project.

--- Note ---
Autobound projects might not get priority:
- if they are subprojects in a Container project, see Working with Container projects on page 11-105 for details
- if they are opened by the workspace, see Projects in workspaces on page 10-6 for details.

---

**Adding Specific_device settings**

The Specific_device setting defines an ordered device list to which a project can autobind. For example an ARM9 core is the first choice but an ARM7 core is the second choice.

You can also use partial matching to control how projects autobind to different processors. For example, you might have an ARM966 and ARM926 core. If you specify ARM966 the project only binds to the ARM966 core. However, if you specify ARM9 the project can bind to either the ARM966 or the ARM926 connection. In addition, if you have specified multiple devices where ARM9 appears before ARM966, RealView Debugger attempts to bind the project using the ARM9 device first. Therefore, the project always binds to an ARM9 core.

This matching behavior means that, where you have specific device names and a partial device name, it might be useful to place the partial device name at the end of the list, as a catch-all binding. See Changing the order of Specific_device settings on page 11-94 for details on how to change the order of devices.

To specify autobinding:

1. Select **Project → Project Properties**... to display the Project Properties window.
2. Select the *PROJECT group in the List of Entries pane.
3. Right-click on the Specific_device setting in the Settings Values pane and select **Edit Value** from the context menu.

4. Enter the specific device name or family name (for example **ARM7TDMI** or **ARM7**), then press Enter to complete the entry.
   A new setting is created, for example `Specific_device ARM7TDMI`. Each device you add creates another entry.

5. Select **File → Save Changes** to save your changes. The makefiles for the project are regenerated.

6. Select **File → Close Window**.

If the project is already bound then changing the Specific_device settings value does not reset the binding. RealView Debugger only attempts to autobind when you open a project. Therefore, after changing the Specific_device settings value either:

- reopen the project to see the change
- use the Project Control dialog box to rebind the project (see **Forcing binding** on page 11-97 for details).

**Note**

To restore the setting, right-click on the Specific_device setting in the Settings Values pane and select **Delete** from the context menu.

---

### Changing the order of Specific_device settings

If you have multiple Specific_device settings values, RealView Debugger attempts to autobind the project in the order they appear in the settings list. You can change the order using the Settings: List Manager dialog box:

1. Select **Project → Project Properties**... to display the Project Properties window.

2. Select the `+PROJECT` group in the List of Entries pane.

3. Right-click on the Specific_device setting in the Settings Values pane and select **Manage List...** from the context menu to display the Settings: List Manager dialog box.

4. Select the device that you want to move.

5. Click the **Move Up** or **Move Down** button to move the device to the required position in the list.

6. Click **OK** to close the Settings: List Manager dialog box.
7. Select **File → Save Changes** to save your changes. The makefiles for the project are regenerated.

8. Select **File → Close Window**.

---

**Note**

When setting a `Specific_device` value, names must match device names stored in the corresponding JTAG configuration file. See the chapter describing configuring custom targets in *RealView Debugger v1.6 Target Configuration Guide*.

---

**Working with multiple projects**

If you are licensed to work in multiprocessor debugging mode, autobinding is especially useful where there might be binding ambiguity. For example, if you are working with a debug target incorporating two ARM processors, an ARM7 core and an ARM9 core, you can create two projects where one project is autobound to the first device (ARM7), and the second project is autobound to the second device (ARM9). Autobinding enables you to guarantee that the two projects bind correctly.

---

**11.12.2 Viewing project binding**

If you create or open a user-defined project without previously connecting to a debug target, then the project is unbound. This also applies if you create or open a project, and the specified processor family for that project does not correspond to the current connection.

For example, if you connect to an Oak DSP debug target, and you then create a user-defined project for an ARM processor, this new project is shown as unbound in the default Code window title bar, for example:

```
RVDEBUG<ARM-Project_1> = @SimOAK_1:Sim [Unattached]
```

If you are connected to a target when you create or open a user-defined project, RealView Debugger binds the project to the connection if it can. When you load an image to create an in-memory auto-project, or if RealView Debugger opens the saved auto-project associated with the image, the auto-project binds automatically.

The Code window title bar shows that the project is bound to a connection by including the project name in parentheses, for example:

```
RVDEBUG(ARM-Project_2) = @ARM7TDMI_0:ARM-A-RR [Unattached]
```
**Note**

For an auto-project, the name in the title bar is not the image name, but the project name created from the image name.

The title bar updates if you change the project environment, for example if you:

- use the Project Control dialog box to unbind or rebind a project
- use the Project Control dialog box to change the active project
- disconnect.

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.

Use the Project Control dialog box to see the open project list and how projects are bound. See the example in Figure 11-15 on page 11-81.

**Viewing autobound projects**

The Code window title bar shows if a project is bound to the current connection. The title bar does not show if the project is autobound. If you do not want to examine the project properties, you can use the Project Control dialog box to see if a project has specific device binding set.

Any open projects that are autobound have (DEV) appended to the entry in the open project list, shown in Figure 11-20.

![Figure 11-20 Autobound projects in the Project Control](image)
In Figure 11-20 on page 11-96, there are two projects in the open project list. The last project to open is at the top of the list and is selected by default. The dhrystone project is autobound to the current connection, ARM7TDMI_0:ARM-A-RR. You can see that the project is autobound because (DEV) is appended to the entry to show that the project has specific device binding set.

In this case, dhrystone is the active project because it was the last project to bind successfully to the current connection.

--- Note ---
If the dhrystone project is unbound, the entry changes to dhrystone - <none> (DEV).

11.12.3 Forcing binding

Select Project → Project Control... from the default Code window main menu to display the Project Control dialog box where you can force project binding and so change your project environment.

--- Note ---
See Working with the Project Control dialog box on page 11-82 for details on other controls in this dialog box.

In Figure 11-15 on page 11-81, there are three projects in the open project list. The last project to open is shown at the top of the list and is selected by default. The dataabort project is bound to the current connection, using default binding. The dhrystone project is open but unbound.

In this case, dataabort is the active project because it was the last project to bind successfully to the current connection. This means that you can access the project properties from the Project menu or from the Process Control pane, if this is visible.

To bind the dhrystone project to the current connection:

1. Select the interworking project so that it is unticked.
2. Select the dhrystone project so that it is ticked. This selects the project for action.
3. Click Re-Bind to force RealView Debugger to bind the selected project. This displays a list selection box showing the current connection, shown in Figure 11-21 on page 11-98.
Managing Projects

![Figure 11-21 Connection selection box](image)

If you are licensed to work in multiprocessor debugging mode, this list shows all connections that the chosen project could bind to.

4. Select the connection so that it is ticked. You must do this even where the list contains only one connection.

5. Click **OK** to confirm your choice and close the selection box. Otherwise, click **Cancel** if you want to abort the binding operation.

   This updates the contents of the Project Control dialog box to show the new binding.

6. Click **Close** to close the Project Control dialog box.

See **Effects of binding** on page 11-99 for what happens when a project binds.

**Autobound projects**

You can use the Project Control dialog box to force binding for autobound projects. If, for example, the open project list contains four projects where three have a `Specific_device` entry set and one uses default binding, you can select all four projects and then click **Re-Bind** to force binding. Each project is then bound in turn. You can, therefore, use this dialog box to force default binding to displace an autobound project.

You cannot use the Project Control dialog box to force an autobound project to bind to a device that does not match. You must change the `Specific_device` setting first, see **Adding Specific_device settings** on page 11-93 for details.

**Forcing unbinding**

You do not have to unbind a project before you rebind another project to the same connection. However, you might want to unbind a project so that you can open a new project and ensure that it binds to the connection, for example if you have an autobound project bound to the current connection and you want to open a project that uses default
binding. Use the Project Control dialog box to unbind projects. Do this in the same way as rebinding but click **Un-Bind** to unbind the selected project(s). See *Effects of unbinding* on page 11-100 for what happens when a project unbinds.

If you unbind a user-defined project where the image associated with that project is loaded to the debug target, RealView Debugger tries to use an auto-project to store settings for the image. If you then open and bind another project to this connection, RealView Debugger displays both projects in the Process Control pane. In this case, the Project Control dialog box shows three projects:

- the auto-project associated with the current image
- the second user-defined project bound to the current connection
- the user-defined project opened first.

See *Managing multiple projects* on page 11-102 for details on working with multiple projects in this way.

### 11.12.4 Effects of binding

When a project binds to a connection:

- Any load-related commands specified in the **SETTINGS** group are executed, for example loading the image, setting top of memory, or semihosting.

  As a minimum, binding registers the image details with RealView Debugger. This means that the debugger can autoload the image on a **GO** or **RELOAD** command. See **SETTINGS group** on page B-9 for more details.

- Any open commands you have specified are executed. These are defined in the project settings file in the **Command_Open_Close** group in the top-level **PROJECT** group. See **Command_Open_Close group** on page B-6 for more details.

  RealView Debugger also executes these commands if you use the Project Control dialog box to rebind the project (see *Forcing binding* on page 11-97).

- RealView Debugger updates the default Code window title bar.

- If the Process Control pane is visible, the display in the **Process** tab is updated with project details.

- If you force a project to bind to a connection and there is already a project bound to that connection, the first project unbinds. This applies even where the first project is autobound. See *Effects of unbinding* on page 11-100 for details on what happens when the first project unbinds.
11.12.5 Effects of unbinding

When a project unbinds from a connection:

- RealView Debugger updates the default Code window title bar.
- If the Process Control pane is visible, then all process details are removed from the Process tab.
- Any close commands you have specified are executed. These are defined in the project settings file in the Command_Open_Close group in the top-level PROJECT group. See Command_Open_Close group on page B-6 for more details. RealView Debugger also executes these commands if you use the Project Control dialog box to unbind the project (see Forcing unbinding on page 11-98).

11.12.6 Connecting and disconnecting

Connecting to and disconnecting from a debug target changes the project environment:

- If you open a project and then connect to a target, RealView Debugger tries to bind the project to the connection in the normal way.
- If you open multiple projects and then connect to a target, RealView Debugger binds any autobound projects first. Default binding is used where there are no autobound projects.
- If you connect to a target, this might change the active project depending on the current project environment.
- If you connect to a target, RealView Debugger updates the default Code window title bar.
- If you connect to a target and a project binds, RealView Debugger updates the Process Control pane, if this is visible.
- If you connect to a target and a project binds, any open commands you have specified as part of the project definition are executed.
- If you disconnect, this might change the active project depending on the current project environment.
- If you disconnect, RealView Debugger updates the default Code window title bar.
- If you disconnect, RealView Debugger clears the Process Control pane.
- 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, 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.

See Connecting and disconnecting on page 11-110 for details on what else happens if you are working with multiple projects and you connect or disconnect.
11.13 Managing multiple projects

This section describes how to work with multiple projects in a debugging session. It provides hints and tips if you are working with:

- multiple user-defined projects
- Container projects
- multiple auto-projects
- mixed user-defined projects and auto-projects.

It assumes that you are familiar with the concepts and terms described in all the previous sections in this chapter.

Note

If you are licensed to work in multiprocessor debugging mode, read this section for an introduction to working with multiple projects across different connections.

This section describes:

- Working with multiple projects
- Active projects on page 11-103
- Working with Container projects on page 11-105
- Working with auto-projects on page 11-107
- Closing projects on page 11-109
- Connecting and disconnecting on page 11-110.

11.13.1 Working with multiple projects

When you are working with a single project, it is normally bound to the connection and so you can perform any action on it easily.

Where you are working with multiple projects, the project environment depends on:

- the order in which projects open
- project binding
- any loaded images.

If you are working with a single user-defined project, where the image associated with that project is loaded to the debug target, and you unbind the project, RealView Debugger tries to use an auto-project to store settings for the image. If you then rebind the user-defined project, it replaces the auto-project in the Process Control pane.
Note

It is recommended that you do not do this. Always use the user-defined project for an image that you load, where this exists. Only use an auto-project for an image where no user-defined project exists.

If you unbind a user-defined project where the image associated with that project is loaded to the debug target, RealView Debugger uses an auto-project to store settings for the image. If you then bind a different project to this connection, the Process Control pane shows details of both projects. See Working with auto-projects on page 11-107 for details.

Therefore, when you are working with several open projects, remember:

- By default, the project shown in the default Code window title bar is the active project. If you are not connected to a debug target, this defaults to the last project you opened. If you are connected, this is the last project to bind successfully. See Active projects for more details on active projects.

- Project-level commands selected from the Project and Tools menus act on the active project only.

- The Process tab, in the Process Control pane, shows details about the bound project and gives you access to the project properties. This can also be used to load and unload the image associated with the project.

- If you are connected, an open project that is unbound is usually not the active project. This means that you cannot access the project properties from the Project menu, or from the Process Control pane.

Use the Project Control dialog box to work on an unbound project without changing the project environment, for example where you do not want to rebind to a connection. See Using the Project Control dialog box on page 11-81 for details.

- The Process Control pane might show details for more than one process depending on the properties of the project. See Working with auto-projects on page 11-107 for more details.

11.13.2 Active projects

When working with multiple projects, the active project is the project that you can modify or build in RealView Debugger:

- There can be only one active project for a connection.

- The active project name appears in the default Code window title bar.
Managing Projects

- The active project is usually bound to the connection.
- The project base directory for the active project usually defines the current working directory.

You can perform any action on this project. However, you might have to perform actions on other projects that you have open. RealView Debugger enables you to change the project shown in the Code window title bar, so that you can perform actions on a different project. That is, you can make another project the active project. See Changing the active project for more details.

**Changing the active project**

In Figure 11-15 on page 11-81, there are three projects in the open project list. The last project to open is shown at the top of the list and is selected by default. The dataabort project is bound to the current connection, ARM7TDMI_0:ARM-A-RR, using default binding. The dhrystone project is open but unbound.

In this case, dataabort is the active project because it was the last project to bind successfully to the current connection. This means that you can access the project properties from the Project menu or from the Process Control pane, if this is visible.

---

**Note**

If you are licensed to work in multiprocessor debugging mode, you can specify an active project for each active connection. This is in addition to the default active project. See the chapter describing multiprocessing in RealView Debugger v1.6 Extensions User Guide for more details.

---

To change the active project:

1. Close any open source files in the File Editor pane. This is to ensure that you do not have any files open that are for other projects if you use the Project menu to execute project-level commands, for example adding source files.
2. Select Project → Project Control... from the default Code window main menu, to display the Project Control dialog box.
3. Click the check box for the current active project so that it is not checked.
4. Click the check box for the project that you want to make the active project.
5. Click Active.

The project entry is moved to the beginning of the open project list. An asterisk (*) is placed before the project entry to show that the active project has changed from the default.
The default Code window title bar changes to show the new active project and that the project is unbound.

6. Click **Close** to close the Project Control dialog box.

--- **Note**

Project binding is not affected when you change the active project in this way.

If you select multiple projects from the list and then click an action control, the specified action is carried out on the selected projects in the order they are listed and so, in some cases, the last action completed successfully overwrites all previous actions. For example, if you have three projects open, deselect the first project in the list, select the last two projects, and then click **Active**. This makes the last selected project the active project, and RealView Debugger moves that project to the top of the list.

**Reactivating a project**

If you click the **Active** button again for a project that you have previously set to be the active project, the asterisk is removed from the project name in the Project Control dialog box.

If there is a project already bound to the current connection, this project becomes the active project. If there is no project bound to that connection, then the project that you previously made active, remains the active project.

**11.13.3 Working with Container projects**

A Container project uses only a subset of the project settings. The project settings file contains a single group, the **PROJECT** group, that holds a list of **subprojects**. When you open a Container project, RealView Debugger opens the associated subprojects, shown in Figure 11-22 on page 11-106. This means that, by definition, you are working with multiple projects.

For a Container project, the order of the subproject list defines the order in which the subprojects open and build. You must specify the subproject list, therefore, in order of dependency.

See **Steps for creating a Container project** on page 11-33 for details on how to create a Container project.

If you open a Container project:

- The project binds to the current connection using the normal default binding rules (see **Types of binding** on page 11-90 for details).
Managing Projects

- All subprojects open but do not bind, even where they are autobound. RealView Debugger warns of any subprojects it cannot open.
- The open project list is updated to show all the open projects, including subprojects.
- The Container project becomes the default active project for the connection.

--- Note ---

A Container project cannot be autobound but subprojects, held in the PROJECT group, can define one or more specific device binding settings. Therefore, a Container project cannot open and bind to a connection if there is an autobound project already bound to that connection.

---

Working with the Project Control dialog box

You can perform actions on one or more of the subprojects that make up a Container project from the Project Control dialog box, shown in Figure 11-22, for example add new source files. You can access project properties for any subproject or control subproject binding.

![Project Control](image)

**Figure 11-22 Container projects in the Project Control**

In Figure 11-22, there are five projects in the open project list. The active project is shown at the top of the list and is selected by default. This has been changed to be the default active project, see *Changing the active project* on page 11-104 for details.

The Container project is bound to the current connection, ARM7TDI_0:ARM-A-RR. It defines two subprojects, shown by the plus sign appended to the entries. Both are open but unbound. In this example, the two subprojects are not Container projects.
You cannot add files to a Container project, only to a subproject. To do this, use the Project Control to make the subproject active. For details see:

- Changing the active project on page 11-104
- Adding files to a user-defined project on page 11-23.

If you close a Container project, all subprojects close automatically. However, you can close one or more subprojects while keeping the Container project open. If you try to build a Container project where one of the subprojects is closed, this operation fails.

--- Note ---
The Container project defines the current working directory. This defines relative pathnames when you make changes to subprojects within the Container project.

Nesting Container projects

Container projects can be nested but not recursive, that is a Container project can include other Container projects but must not include itself. However, the nested Container project has no makefile and so any build fails.

If you open a Container project that defines a Container subproject, the Project Control dialog box shows a second level in the hierarchy.

Do not open a nested Container project if one of the subprojects is already open. Close the component project first so that the display is consistent.

11.13.4 Working with auto-projects

Any image that you load to a debug target must be accompanied by a project to save the image-related settings. If you open a user-defined project and this binds to a connection then this can be used to load the associated image. If you load an image directly, RealView Debugger searches for a saved auto-project file in the same location. If a saved file does not exist then RealView Debugger creates an in-memory auto-project to use in this session. You can save this file for later use.

If you load multiple images to the same target processor, the Process Control pane shows entries for the images and the auto-projects used to hold image-related settings, see Working with multiple images on page 2-17 for details on loading two images.

This means that you can work with multiple auto-projects or mix auto-projects and user-defined projects in the same session:

- If you are working with multiple projects in this way, normal binding rules apply, see Types of binding on page 11-90 for details.
Managing Projects

- If you display the Project Control dialog box, the open project list contains both auto-projects and user-defined projects. You can now work on both types of project in the usual way, see Working with the Project Control dialog box on page 11-82 for details.

- It is not possible to distinguish between auto-projects and user-defined projects in the Project Control dialog box where they have the same name, unless the user-defined project is autobound. When you close one of the projects, use the Process Control pane to guarantee that you close the correct project, if possible. See Managing projects in the Process Control pane on page 11-84 for more details on these operations.

RealView Debugger enables you to view a combination of image details, auto-projects and user-defined projects in the Process Control pane, shown in Figure 11-23.

![Figure 11-23 Mixing projects in the Process Control pane](image)

Figure 11-23 Mixing projects in the Process Control pane

Figure 11-23 shows:

- Two, non-overlapping, images loaded to the target processor.
- The first image, hello.axf, is associated with an in-memory auto-project.
- The second image, dhrystone.axf, is associated with a saved auto-project.
- The project settings file interworking.prj is open and bound to the current connection.
- The default active project is interworking.prj, shown in the default Code window title bar.
However, the Project Control dialog box, shown in Figure 11-24, indicates that there are two more user-defined projects open in this session.

![Project Control dialog box]

**Figure 11-24 Mixing projects in the Project Control dialog box**

Figure 11-24 shows, from the top:

- The dhrystone auto-project associated with the loaded image. This is unbound.
- The user-defined project interworking bound to the current connection.
- The dataabort user-defined project. This is open and unbound.
- The dhrystone user-defined project. This is open and unbound. This project defines specific device binding.
- The hello auto-project associated with the loaded image. This is unbound.

This example demonstrates the following rules:

- Unbound user-defined projects are not visible in the Process Control pane.
- If a user-defined project is bound to the connection, and you manually bind another user-defined project to the connection, then the process details for the second project replace the process details of the first project, in the Process Control pane.
- Auto-projects, associated with loaded images, are always visible in the Process Control pane, even where the auto-project is not bound to the connection.

### 11.13.5 Closing projects

When you are working with multiple projects, you can close one or more projects as follows:

1. Select **Project → Close Project** to see the open project list.
2. Select the project(s) that you want to close.

3. Click **OK**.

If you close a project that is bound to a connection, RealView Debugger uses the open project list to bind another project to the connection. The normal binding rules apply, see *Types of binding* on page 11-90 for details.

If you close a bound project that is also the active project, RealView Debugger sets the default active project:

- The next project that binds to the connection becomes the active project.
- If no project binds to the connection, the active project is the first project in the open project list.

If you close an unbound project that is also the active project, then the next project in the open project list becomes the active project by default.

--- **Note**

When you close a project that is bound, RealView Debugger executes any close commands specified in the **Commands_Open_Close** group. See *Command_Open_Close group* on page B-6 for more details on the available commands.

---

### 11.13.6 Connecting and disconnecting

Connecting to and disconnecting from a debug target changes the project environment:

- If you open multiple projects and then connect to a target, RealView Debugger binds any autobound projects first.
- If you open multiple projects where none specifies autobinding and then connect to a target, RealView Debugger uses default binding to bind projects in the order specified by the open project list.
- 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. This clears the Process Control pane.
- If you disconnect, 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.
Chapter 12
Editing Source Code

This chapter describes the file editing facilities that RealView Debugger provides. It contains the following sections:

- About the File Editor on page 12-2
- Configuring the File Editor on page 12-4
- Using the Editor Buffer List on page 12-6
- Editing text on page 12-9
- Managing your editing session on page 12-15
- Using standalone editors on page 12-19
- Using templates on page 12-21
12.1 About the File Editor

The File Editor provides a range of code editing features and enables you to specify your own personal editor if required. The File Editor enables you to:

- enter text to create project files
- open source files for editing and resaving
- edit binary files and save in a specified format
- use a range of text searching operations
- use full emulation of vi, ex, Visual C++, and Brief.

**Caution**

RealView Debugger limits the way the File Editor handles binary files. See Editing binary files on page 12-3 for important information.

This section describes:

- Using the File Editor pane
- Using standalone editors on page 12-3
- Using standalone editors on page 12-3
- Editing binary files on page 12-3.

### 12.1.1 Using the File Editor pane

The File Editor pane enables you to perform basic editing operations. Figure 12-1 shows two files loaded into the File Editor and displayed in the File Editor pane.

![File Editor pane](image)

Figure 12-1 File Editor pane

Line numbers are not displayed by default but, in this example, this feature has been enabled. See Changing your Editing Controls on page 12-4 for details of how to set this up for the current session.
The File field shows the name of the file currently displayed in the File Editor pane. If a file is *dirty*, that is if you have changed the file since loading or saving, an asterisk (*) is appended to the end of the filename. Hold your mouse pointer over this field to see the full pathname.

The **Source control** button indicates the read/write status of the current file. Click this button to change the status of a file loaded into the File Editor. You can only edit the file if the Read-Write icon is displayed.

As each file is loaded into the File Editor, it creates a file tab in the File Editor pane. Click on the file tabs at the bottom of the File Editor pane to choose which file is currently visible. If a file has been edited, an asterisk (*) is appended to the front of the filename.

### 12.1.2 Using standalone editors

You can use the File Editor outside a Code window by creating new standalone File Editor panes using the **Tools** menu. See *Using standalone editors* on page 12-19 for full details on using these features.

### 12.1.3 Using personal editors

When working with source files, RealView Debugger enables you to use a personal editor to create source code or to modify existing files. See *Using standalone editors* on page 12-19 for full details on using these features.

### 12.1.4 Editing binary files

Be aware of the following warnings if you are editing binary files in the File Editor:

- Any binary file opened into the File Editor is truncated to 256KB without warning.
- If you open a binary file into the File Editor, the **Save As...** option is enabled. Selecting this option saves the truncated file.
- If you edit a binary file, the **Save** option is disabled. This means that your original file is safe from accidental corruption.
- The display format used in the File Editor pane means that you can edit any part of the file, including spaces. Be careful when saving a binary file that you have changed.
12.2 Configuring the File Editor

You can configure how to display files in the File Editor:

- *Changing your Editing Controls*
- *Working with multiple files.*

12.2.1 Changing your Editing Controls

Select **Edit → Editing Controls** from the main menu to set up temporary File Editor display options. This displays the **Editing Controls** menu shown in Figure 12-2.

![Figure 12-2 Editing Controls menu](image)

Select options from this menu to set:

- the default size for all tabs in the text
- the character width for Shift Lines Left and Shift Lines Right
- the text coloring scheme for source code
- automatic line numbering and renumbering
- tooltip evaluation for variables and registers.

Changes to line numbering and tooltip evaluation apply to the current Code window and do not persist to any sibling windows. However, enabling text coloring or changing the default tab size or character width (for Shift Lines Left and Shift Lines Right) applies to all Code windows in the current session.

For more details on configuring your editing environment see *Managing your editing session* on page 12-15.

12.2.2 Working with multiple files

You can load source files into the File Editor using the menu option **File → Open....** Multiple filenames can be highlighted in the File Chooser dialog and each file is loaded into its own tab.

You can move files between File Editor panes using drag-and-drop. If there is only a single file in the source window, performing the drag closes this window automatically.
Similarly, you can move files between Code windows using drag-and-drop. If there is only a single file in the source window, performing the drag closes this window automatically. This does not apply if the source window is the default Code window, as closing this window ends your debugging session.

With files loaded into the File Editor, you can create an empty tab. To do this, select **File → New → Text File** from the Code window main menu. This creates an empty tab, named *<None>* , ready to enter text from the keyboard or to paste text from another file.

**Using standalone File Editor windows**

You can set up multiple File Editor windows outside the current Code window to give you access to all the built-in editing features. With at least one file loaded into the File Editor, click on the file tab for the required file and select either:

- **View → Editor Window**
- **Tools → New Editor → New Editor Copy.**

This displays a standalone editor window and automatically loads the selected file into it. This is useful for looking at different parts of the same file simultaneously.

In the same way, select **Tools → New Editor → New Editor Empty** to display an empty File Editor window and the File Chooser dialog from where new files can be selected and loaded.

**Using the Editor Buffer List**

You can use the Editor Buffer List to start a second instance of the File Editor:

1. Select **Tools → New Editor → Editor Buffer List...** to see a list of active files.
2. Select a file from the list of previously used files.
3. Click **Edit** to display a new File Editor pane containing the chosen file.
12.3 Using the Editor Buffer List

The Editor Buffer List, enables you to:

- manage your files in the File Editor
- see a list of files currently active in the File Editor
- view details of previously used files
- organize your desktop during a debugger or editor session.

Select **Tools → New Editor → Editor Buffer List**... to display the Editor Buffer List, shown in Figure 12-3.

![Editor Buffer List](image)

Figure 12-3 Editor Buffer List

The Editor Buffer List dialog box includes a File List holding up to 32 items. This example shows a list of Active Files and Previously Used Files. One file is running in two instances of the File Editor, shown by the 2 before the filename. An asterisk before the filename also shows which files are dirty. The last file loaded into the File Editor is shown at the top of the Active Files list.

Click on a file tab at the bottom of the File Editor pane and select **Close**. This changes the status of the file in the File List. The file tab is deleted. Click **ReScan**, on the Editor Buffer List dialog box, to move the file from the Active Files list to the Previously Used Files list.

Similarly, click on a file tab and select **Close File, Keep Tab** from the file tab context menu to move the file from the Active Files list to the Previously Used Files list. However, this does not remove the tab. Instead a plus sign is appended to the filename on the tab indicating that this file must be re-opened for editing to continue. This saves on memory but also means that a file can be edited in another editor and reopened in the File Editor by clicking on the tab.
The Editor Buffer List dialog box contains:

- **AllOn**: Selects all entries in the File List as indicated by a check in the accompanying check box.
- **AllOff**: Unselects all entries in the File List as indicated by no check in the accompanying check box.
- **Minimize**: Select a file, or files, in the Active Files list and click this button to minimize all associated windows down to the Windows Taskbar.
- **Restore**: Select one or more files in the Editor Buffer List and click this button to restore all associated windows to the desktop.
- **Save**: Enables you to save a file that has been modified. Select all the files in the display list to save all your working files in one step.
- **Edit**: Select a file in the Previously Used Files list and click **Edit** to reactivate a file. The file is opened for editing in a standalone File Editor window.
- **ReScan**: Click this button to refresh the file list, and the status of each file.
- **Close**: Closes the Editor Buffer List dialog box.
- **Help**: Displays online help.

### 12.3.1 Displaying files in the Editor Buffer List

You do not have to close the Editor Buffer List during your editing session as this might help with organizing your files. However, the File List must be kept up to date, using **ReScan**, to ensure that files can be accessed.

To demonstrate file operations:

1. Load two files into the File Editor, for example `sample_arm.c` and `dhry_1.c`.
2. Select **Tools** → **New Editor** → **Editor Buffer List**....
   The Active Files list shows:
   ```
   --- Active Files ---
   1 d:\program files\arm\realview debugger\...dhrystone\dhry_1.c
   1 d:\program files\arm\realview debugger\...sample\sample_arm.c
   --- Previously Used Files ---
   Do not close the Editor Buffer List.
   ```
3. Right-click on the `sample_arm.c` file tab in the File Editor pane and select **Close**.
   The Active Files list shows:
--- Active Files ---
1 d:\program files\arm\realview debugger\...dhrystone\dhry_1.c
0 d:\program files\arm\realview debugger\...sample\sample_arm.c
--- Previously Used Files ---
4. Select the sample_arm.c entry in the Editor Buffer List and click Edit. This results in an error message to say that the file is no longer available.

5. Click ReScan to refresh the File List.
The Active Files list shows:
--- Active Files ---
1 d:\program files\arm\realview debugger\...dhrystone\dhry_1.c
--- Previously Used Files ---
0 d:\program files\arm\realview debugger\...sample\sample_arm.c

6. Select the sample_arm.c entry and click Edit. This displays a standalone File Editor window ready for editing. The Active Files list is updated to show the change.
12.4 Editing text

The File Editor provides full editing options:

- **Basic text editing**
- **Formatting text** on page 12-11
- **Undoing changes** on page 12-13

12.4.1 Basic text editing

The File Editor supports the standard Windows editing operations.

**Opening files**

You can open saved files ready for editing in the File Editor:

- Select **File → Open...** from the Code window main menu. This displays a Select File to Open dialog box where you can locate the required file.

**Closing files**

You can save files after editing, for example:

- Select **File → Save** from the Code window main menu. This saves the current file using the original filename.

- Select **File → Save/Close Multiple...** from the Code window main menu. Select from the file list to save several files in a single step, shown in Figure 12-4.

![Figure 12-4 Save Multiple dialog box](image)

The file displayed in the topmost tab of the calling window is checked by default ready to save. If a file has been edited, an asterisk (*) is appended to the front of the filename.
Select the files that you want to work on, or click AllOn to select all the files:

**Close File**  Closes selected files and removes their tabs from the File Editor pane. This closes the Save Multiple dialog box.

**Save**  Saves selected files and closes the dialog box.

**Convert**  Converts files from DOS to UNIX by removing end-of-line markers, or adds markers to convert UNIX files to DOS format:

- DOS files contain \r (carriage return) and \n (line feed)
- UNIX files contain \n (line feed).

Also converts a binary file to a text file (in the local format).

The Type column shows the current file type.

**Cancel**  Closes the Save Multiple dialog box without making any changes to the files.

**Help**  Click to display online help.

RealView Debugger always warns if you select a file for closing that has been edited.

### Adding text

To add text to an open file:

1. Click once in the File Editor pane to position the text insertion point.
2. Begin typing on the keyboard to enter text.

You can insert text from another file by copying the text block to the clipboard and pasting into your source file at the required position.

In addition, you can use a template to insert text into an open source file within the File Editor. See Using templates on page 12-21 for information.

--- **Note** ---

RealView Debugger warns if the destination file is read-only and asks for confirmation to enable you to edit the file.

### Creating text

To add text to a new file:

1. Select **File → New → Text File** from the Code window main menu. This creates an empty tab, called <None>, in the File Editor pane.
2. Begin typing on the keyboard to enter text.

3. Right-click on the tab to display the File tabs context menu.

4. Select Save File to specify the new filename and location.

You can insert text from another file by copying the text block to the clipboard and pasting into your source file at the required position.

In addition, you can use a template to insert text into an open source file within the File Editor. See Using templates on page 12-21 for information.

Deleting text

You can delete text in any of the following ways:

- Press the Backspace key to delete text that precedes the text insertion point.
- Press the Delete key to delete text that follows the text insertion point.
- Select the text you want to delete and press the Backspace or Delete key to delete the selection.

Note

RealView Debugger warns if the file is read-only and asks for confirmation to enable you to edit the file.

12.4.2 Formatting text

Where a file can be edited, you can format your source code to:

- shift text left or right
- change case
- change capitalization
- enable source code coloring.

Use the Edit menu to set these options for the current session. To make the new setting the default when starting up the File Editor, you must change the Edit settings in your workspace.

For more details on formatting text see:

- Shifting text left or right on page 12-12
- Changing case on page 12-12
- Enabling source code color on page 12-12.
**Shifting text left or right**

To shift blocks of text left and right:

1. Select a block of text.
2. Select Edit → Format to display the Format menu.
3. Select either Shift Lines Left or Shift Lines Right as required.

The File Editor shifts the selected text a designated number of characters to the right or left by inserting or deleting spaces at the beginning of every line in the selection. The amount that text is shifted is determined by the value specified for the Shift Width... setting on the Editing Controls menu.

**Changing case**

To change the case of selected blocks of text:

1. Select a block of text.
2. Select Edit → Format to display the Format menu.
3. Select either abcd → ABCD, to change all text to uppercase, or ABCD → abcd, to change all text to lowercase.

**Enabling source code color**

When working in the File Editor you can color source code as an aid to readability. This feature is enabled by default when you first start RealView Debugger. If source coloring is disabled, all text is the same color, usually black.

Table 12-1 shows the color scheme used if text coloring is enabled.

<table>
<thead>
<tr>
<th>Name</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers</td>
<td>Blue</td>
</tr>
<tr>
<td>Strings</td>
<td>Lime green</td>
</tr>
<tr>
<td>Keywords</td>
<td>Red</td>
</tr>
<tr>
<td>Comments</td>
<td>Gray background</td>
</tr>
<tr>
<td>User defined keywords</td>
<td>Yellow</td>
</tr>
</tbody>
</table>
Standard C and C++ source coloring is auto-enabled based on file extension. The default list is defined in the Edit settings in your workspace and can be changed if required. Specify file extensions as a comma-separated list which must not include dots or periods, for example c, cc, cxx, cpp, h,hpp.

12.4.3 Undoing changes

The File Editor provides several ways to undo mistakes as you edit a file.

**Undoing edits**

Select Edit → Undo to reverse the effect of your last action. The number of levels of undo is defined by the Undo value in the Edit settings in your workspace. The default value gives 64 levels of undo.

**Undoing and redoing edits**

Select Edit → Redo to redo your last undo. The number of levels of redo is defined by the Undo value in the workspace Edit settings.

**Reverting to the last saved version of a file**

You can discard all changes you have made since the last time you saved your file. Select Re-Open File from the file tabs context menu to return a file to the last-saved version. Because the current file is dirty you are warned to save changes before reloading.

12.4.4 Using drag-and-drop

You can use drag-and-drop to move selected text within the File Editor pane or to move text between File Editor panes.

A grayed-out box appears during the drag to represent the text being moved. The top-left corner of this box represents the starting point when text is inserted.

If you want to copy the text block rather than reposition it, hold down the Ctrl key as you drag the block. The grayed-out box is accompanied by a plus sign to show that this is a block copy.

Enabled by default, drag-and-drop text editing can be disabled in your current workspace. See Changing your workspace settings on page 12-17 for details of how to do this.
Moving text between windows

To drag and drop text between instances of the File Editor you must have:
• two or more instances of the File Editor open on the desktop
• files open in each instance of the File Editor
• read/write access to any file into which you are moving text.

If the file that you are moving text from is set to read-only then the selected text is copied into the second window regardless of the action of the Ctrl key during the drag.

Moving files between windows

You can use drag-and-drop to move files between File Editor panes by clicking on the required file tab and dragging it into a new File Editor pane and dropping it over the File Editor pane.

If you are moving a file and it is the only file in the File Editor, the source window is closed when the file has been transferred.

Note

RealView Debugger opens a file as appropriate for the defined file type. If the file is an executable, for example htree.axf, it is loaded to the connected target automatically.
12.5 Managing your editing session

When you are working in the File Editor, there are different ways to configure your working environment:
- *Using the Find menu*
- *Using Editing Controls*
- *Setting source search paths on page 12-17*
- *Changing your workspace settings on page 12-17.*

12.5.1 Using the Find menu

When editing source files you can obtain information about the current editing session using the **Find** menu from the Code window main menu.

Table 12-2 describes the options available from this menu.

<table>
<thead>
<tr>
<th>Table 12-2 Find menu options</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Option</strong></td>
</tr>
<tr>
<td>Where Am I...</td>
</tr>
<tr>
<td>Show Insert Cursor</td>
</tr>
<tr>
<td>Show Last Changed Line</td>
</tr>
</tbody>
</table>

12.5.2 Using Editing Controls

Select **Edit** → **Editing Controls** from the Code window main menu to change editor settings for the current session.
Table 12-3 describes the options available from this menu.

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show Line Numbers</td>
<td>Defines whether line numbers are automatically displayed during editing. If enabled, lines are reordered as new lines are added or deleted.</td>
</tr>
<tr>
<td>Use Original Line Numbers</td>
<td>If selected, and line numbers have been enabled (see above) line numbers are not changed as a file is edited. So, if new lines are added they have no numbers, and if lines are deleted there are gaps in the source file.</td>
</tr>
<tr>
<td>Reset Original Line Numbers</td>
<td>If selected this option immediately resets line numbering to take account of line inserts and deletes.</td>
</tr>
<tr>
<td>Tab Size...</td>
<td>Sets the size of tab stops when editing source code. The default is 8 characters. The maximum value accepted is 16.</td>
</tr>
<tr>
<td>Shift Width...</td>
<td>Sets the number of characters by which text lines are shifted when formatting.</td>
</tr>
<tr>
<td>Auto Indent</td>
<td>Specify auto-indent for a new line when entering source code.</td>
</tr>
<tr>
<td>VI mode</td>
<td>Switches to vi mode for editing source code files.</td>
</tr>
<tr>
<td>Text Coloring</td>
<td>Defines whether text coloring is enabled for source code.</td>
</tr>
<tr>
<td>Tooltip Evaluation</td>
<td>Defines whether tooltip evaluation of variables and registers is enabled.</td>
</tr>
</tbody>
</table>

**Persistence**

If you open a new Code window, RealView Debugger uses editor settings as defined by the current workspace, or the global configuration file if you are working without a workspace. Any changes that you make to these settings in the calling window do not, therefore, persist to the new window.

Exceptions to this rule are:

- Tab Size
- Shift Width
- Text coloring.
12.5.3 Setting source search paths

The File Editor working directory is the location searched for a file when you do not give the full pathname. This is also the first place accessed by the debugger when searching for files.

When you are connected to a debug target, select **Debug → Set Source Search Path...** to specify how the search is configured. When this path has been set up, this becomes the File Editor working directory and is the first place examined for new files. See *Searching for source files* on page 3-16 for details on using this dialog box.

12.5.4 Changing your workspace settings

You can change editor defaults by changing the **Edit** settings in your current workspace or in your global configuration file. Make changes here so that these settings are used the next time RealView Debugger starts and persist to all windows in a session.

If you are working with a workspace, select **Tools → Workspace Options...** to make these changes. If you are working without a workspace, select **Tools → Options...** to make these changes to your global configuration file so that they are available in future debugging sessions.

Table 12-4 describes **Edit** settings that you can change.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backup</td>
<td>File backup controls.</td>
</tr>
<tr>
<td>Tab_conv</td>
<td>Tab to space, and space to tab, conversion settings.</td>
</tr>
<tr>
<td>Src_ctrl</td>
<td>Source control settings.</td>
</tr>
<tr>
<td>Drag_drop_dis</td>
<td>Disable drag-and-drop text editing.</td>
</tr>
<tr>
<td>Vi</td>
<td>If enabled, this will automatically start the File Editor in vi mode.</td>
</tr>
<tr>
<td>Indent</td>
<td>Specify auto-indent for a new line when entering source code.</td>
</tr>
<tr>
<td>Undo</td>
<td>Specify the levels of undo and redo.</td>
</tr>
<tr>
<td>Tab</td>
<td>Specify tab spacing in the File Editor.</td>
</tr>
<tr>
<td>Shift</td>
<td>Specify the number of characters for Shift Left, Shift Right, and auto-indent.</td>
</tr>
<tr>
<td>Line_number</td>
<td>If set to True, this displays line numbers. The default is False.</td>
</tr>
</tbody>
</table>
Table 12-4 Edit settings in workspace (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>No_tooltip</td>
<td>If set to false, the default, this displays hover-style evaluation when you hold your mouse pointer over a variable in the Src tab or a register in the Dsm tab.</td>
</tr>
<tr>
<td>Timer</td>
<td>The File Editor periodically checks to see if another editor has changed the current file. The default value is 60 seconds. You can enter a new value of &gt;30 seconds. Setting this to -1 disables the check.</td>
</tr>
<tr>
<td>Tool_save</td>
<td>Tool-save allows for automatic saving of files in a build. The default is to prompt.</td>
</tr>
<tr>
<td>Startup</td>
<td>The default start-up file rvdebug.sav holds RealView Debugger information and the list of previously used editor files. Setting this to -1 disables this file.</td>
</tr>
<tr>
<td>Template</td>
<td>The default template file, see Using templates on page 12-21.</td>
</tr>
<tr>
<td>Restore_state</td>
<td>The restore state for standalone editor files only, see Using standalone editors on page 12-19.</td>
</tr>
</tbody>
</table>

For full details on changing your workspace settings to define editing controls, see Chapter 10 Configuring Workspace Settings.
12.6 Using standalone editors

You can use the File Editor in standalone mode outside the Code window.

You can start up a standalone editor window in two ways:

- With a file displayed in the File Editor pane, select Tools → New Editor → New Editor Copy from the Code window main menu. This displays a standalone editor window with the current file already loaded ready for editing, shown in Figure 12-5.

- Select Tools → New Editor → New Editor Empty from the Code window main menu. This displays an empty standalone editor window and the File Chooser dialog box where a file can be opened.

When a file has been loaded, the standalone editor provides the same functionality as the File Editor pane except:

- the Debug menu option is no longer available
- the Tools menu has fewer options
- Editing Controls defaults are used
- shortcut buttons are no longer available.

If you are working with a standalone editor, the following notes apply:

- The standalone File Editor pane displays as a floating window that is independent of the calling Code window.
Edits made in the standalone editor window echo in the calling File Editor pane.

Edits made in the calling File Editor pane echo in the standalone editor window.

You can move files between standalone editor windows in the same way as between File Editor panes in the Code window.

You can configure the standalone editor to start up with the last-used file, or files, preloaded. To configure this option you must set the value of `Restore_state` in the `Edit` settings, shown in Table 12-4 on page 12-17.

### 12.6.1 Using a personal editor

RealView Debugger can be configured to work with a personal editor by setting the Windows environment variables `EDIT_CMD`, `EDITOR`, or `VISUAL`. The `VISUAL` variable is checked first.

When set up, select **Tools → New Editor → Personal Editor** to start the chosen editor and open a window to allow file editing.

If you are working with a personal editor:

- The personal editor opens the chosen file from disk and so does not include any changes made since it was last saved.

- The personal editor displays as a floating window that is independent of the calling Code window.

- Edits made in the personal editor window do not echo in the calling File Editor pane.

- Edits made in the calling File Editor pane do not echo in the personal editor window.

If there is no open file in the calling File Editor pane, displaying a personal editor opens an empty window ready for entering text.
12.7 Using templates

Templates are generic segments of text that you can modify to fit particular functions in your source code. You can use the File Editor to create templates or to modify existing templates ready for inserting them into your source code.

12.7.1 Inserting templates

A template can be inserted into an empty File Editor pane or added to existing code. The insertion point is determined by the location of the cursor.

By default, RealView Debugger loads the template file provided as part of the root installation, \etc\rvdebug.tpl, but you can access your own template files if required, see Creating templates on page 12-22 for more information.

To insert a template:

1. Display the file in the File Editor pane.
2. Position the text insertion point at the required location and select Edit → Insert Template... to insert the template.

   This displays the Insert Template dialog box showing the currently available template file and the templates it contains, shown in Figure 12-6.

   ![Figure 12-6 Insert Template dialog box](image)

   If you have created your own template file, use the Load... button to locate and load a different template file, for example my_template.tpl.

3. Highlight the required template in the list and click Show to display a text box showing the contents of the chosen template.

4. Click Close to close the definition display.
5. Highlight the required template in the list and click **Insert** to paste the template into the current source code in the File Editor. The chosen template is displayed and any variables are automatically updated.

6. When all the required templates have been included, close the Insert Template dialog box and save your modified source code.

**Editing templates**

You can edit template files from the Insert Template dialog box, shown in Figure 12-6 on page 12-21. Highlight a template from the list and click **Edit...** to display a standalone File Editor pane already loaded with the specified template file ready for editing.

You can also load a template file directly into the File Editor pane without going through the Insert Template dialog box. You can also create and edit template files using any text editor.

**12.7.2 Creating templates**

You can create your own templates, or modify an existing template, in an editor and then save the newly-created file. When saving your own templates you must use the .tpl extension.

A template takes the format:

```
"templatename" TEMPLATE [-above|-below|-indent|-selection]
  [-auto=<phrase>] [filename]
.
.
.
"templatename" END
```

where:

- `-templatename` Specifies the template name.
- `-above` Use this to insert the template on the line above the current cursor.
- `-below` Use this to insert the template on the line below the current cursor.
- `-indent` Templates are indented to match the previous line.
- `-selection` Indicates that the template contains variables that are replaced on insertion into the source code. Valid variables are:
  - `$selection` The currently selected text.
  - `$paste` The current paste buffer.
$filename

The current filename, without the path, in lowercase.

$FILENAME

The current filename, without the path, in uppercase.

$directory

The path without the filename.

-auto=<phrase>

Specifies the auto insertion phrase. The phrase is a name associated with a defined template. When a template has an auto insertion phrase, there are three actions available after typing in the phrase:

Ctrl+Shift+3  Insert the template at the cursor position.
Ctrl+3  View the definition of the template specified by phrase.
Alt+3  Invoke the Insert Template dialog.

filename

Specifies the file containing text to define the template. Where no file is given, the lines following the template definition header are used until the line "templatename" END is reached.

Within the template, you can place a ^L (ASCII 12) character after a comment section. The lines immediately preceding the ^L are not inserted. These lines are used for data definitions.

You can also nest templates within templates using the syntax:

“filename” INCLUDE

12.7.3 Template examples

Example 12-1 shows a template file using text from the specified file, called myfile.txt. On insertion, the text is placed above the cursor position.

```
# Simple template example using text from a specified file

"TemplateName" TEMPLATE -above C:\RealView Debugger\etc\myfile.txt
"TemplateName" END
```

Example 12-2 on page 12-24 shows a template of a simple FOR loop written in C. On insertion, the text is placed below the cursor position and indented to match the line of C code that immediately precedes it.

```
Example 12-2 on page 12-24 shows a template of a simple FOR loop written in C. On insertion, the text is placed below the cursor position and indented to match the line of C code that immediately precedes it.
```
Example 12-2

```
# Simple template example - simple FOR loop

"Simple For Loop" TEMPLATE -below -indent
for (index = 0; index < top; index++);
{
}
"Simple For Loop" END
```

Example 12-3 shows the use of an auto-insertion phrase. With this template, type the auto-insertion phrase `WriteDevice` and immediately press Ctrl+3 to display all the template text. However, press Ctrl+Shift+3 to insert only the text immediately below the `\^L`.

Example 12-3

```
# Simple template example
# This combines an auto insertion phrase and the use of `\^L` (ASCII 12)

"WriteDevice Function" TEMPLATE -auto='WriteDevice'
EXAMPLE
/* write a message string to the LCD */
if (!WriteDevice(main_lcd, sizeof(init_msg), init_msg))
   ReportFailure(DEV_ERR, class_lcd, main_lcd);
\^L
Status WriteDevice(int dev_no, int len, uint8 *buffer);
"WriteDevice Function" END
```

Example 12-4 shows the use of selection when inserting a template. If you are editing source code and you select a function name within a definition and insert this template, the body is expanded above the function and the function name is filled in where `$selection` is found.

Example 12-4

```
# Simple template example using selection

"Function Header" TEMPLATE -sel -above
/*
---------------------------------
$selection -
Notes:
```
```
Each example can be typed into a .tpl file or they can all be entered into a single file, for example template_examples.tpl, for access through the Insert Template dialog box.
12.8 Working in vi mode

You can use vi as your preferred editor when working on source files in the File Editor. This is a full implementation offering all vi commands. Disabled by default, select Edit → Editing Controls → VI mode from the Code window main menu to switch into vi editor mode. This inserts another command line at the bottom of the File Editor pane where you can enter vi commands.

From here, press Escape to switch between text input mode and command mode. However, when editing your file, mouse actions and menu options can be used interchangeably with vi commands. For example, the yank (y) commands pull text into the paste buffer from where it can be pasted.

Place the mouse over the vi command line to paste commands in the same way as text.

12.8.1 Using ex editor commands

Editing in vi mode also enables you to access several ex commands. These are shown in Table 12-5 and Table 12-6 on page 12-28.

When using ex commands in vi mode, whether using : or / commands, pasted text is applied at the : or / command regardless of the position of the cursor. Similarly, when working in s mode (substitute text at the $ sign), pasted text is applied at the $ sign regardless of the position of the cursor.

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>:a</td>
<td>Append text after cursor.</td>
</tr>
<tr>
<td>:d</td>
<td>Delete current line. Use :.,$d to delete from the current line to the end of the file.</td>
</tr>
<tr>
<td>:e</td>
<td>Exit (warns of unsaved changes).</td>
</tr>
<tr>
<td>:e!</td>
<td>Exit without saving buffer contents.</td>
</tr>
<tr>
<td>:g</td>
<td>Change all occurrences of text on current line.</td>
</tr>
<tr>
<td>:n</td>
<td>Edit next file in buffer.</td>
</tr>
<tr>
<td>:q</td>
<td>Quit (warns of unsaved changes).</td>
</tr>
<tr>
<td>:q!</td>
<td>Quit without saving buffer contents.</td>
</tr>
<tr>
<td>:r file</td>
<td>Read file to specified position.</td>
</tr>
</tbody>
</table>
### Table 12-5 ex commands supported in vi mode (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>:s/str1/str2</td>
<td>Substitute str2 for str1 on current line. Use range to specify line numbers, for example :2,10s/int1/int2 or :1,$s/int1/int2. Use /g to make a global substitution.</td>
</tr>
<tr>
<td>:t</td>
<td>Search and move.</td>
</tr>
<tr>
<td>:w</td>
<td>Write buffer contents. This reports the full pathname, the number of lines and the size.</td>
</tr>
<tr>
<td>:w file</td>
<td>Write buffer contents to file. Edit file specified by file. Use range to specify line numbers to write, for example :2,10w file.</td>
</tr>
<tr>
<td>:x</td>
<td>Kill editor instance. This writes buffer contents and closes the Code window.</td>
</tr>
<tr>
<td>:y</td>
<td>Yank.</td>
</tr>
<tr>
<td>:e file</td>
<td>Edit file specified by file.</td>
</tr>
<tr>
<td>:wq</td>
<td>Write buffer contents and quit.</td>
</tr>
<tr>
<td>:set autoindent</td>
<td>Set indent to match previous line.</td>
</tr>
<tr>
<td>:set sw</td>
<td>Set shiftwidth for soft tabs.</td>
</tr>
<tr>
<td>:set all</td>
<td>Display list of customizing options.</td>
</tr>
<tr>
<td>:set nu</td>
<td>Display line numbers.</td>
</tr>
<tr>
<td>:set nomu</td>
<td>Do not display line numbers.</td>
</tr>
<tr>
<td>:set tabstop</td>
<td>Set tabstop size.</td>
</tr>
<tr>
<td>?</td>
<td>Search backwards in file.</td>
</tr>
<tr>
<td>/</td>
<td>Search forwards in file.</td>
</tr>
<tr>
<td>:</td>
<td>Begin in ex editor.</td>
</tr>
<tr>
<td>!</td>
<td>Execute a shell.</td>
</tr>
</tbody>
</table>
Table 12-6 Additional ex commands supported in vi mode

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>:cwd</td>
<td>Show or set local working directory. Local means local to this window only.</td>
</tr>
<tr>
<td>:cwd*</td>
<td>Set local working directory to that of the current file. Local means local to this window only.</td>
</tr>
<tr>
<td>:nw file</td>
<td>Open file in new standalone editor window.</td>
</tr>
<tr>
<td>:tfunc</td>
<td>Jump to named function. Can also use :tstr* to find a function name starting with str.</td>
</tr>
</tbody>
</table>
Chapter 13
Searching and Replacing Text

This chapter describes how to use RealView Debugger search and replace functions. It contains the following sections:

- About finding and replacing text on page 13-2
- Searching and replacing text on page 13-3
- Searching multiple files on page 13-7
- Working with functions on page 13-10
- Pair matching on page 13-12
- Configuring searches on page 13-15
- Using regular expressions on page 13-17.
13.1 About finding and replacing text

The find and replace functions in the File Editor enable you to:
- work within the current Code window
- work between windows
- work across different windows.

In search and replace operations you can use:
- text strings
- pattern matches
- grep-style regular expressions.

All the search and replace operations described in this chapter are also available when using the File Editor in standalone mode.
13.2 Searching and replacing text

This section describes how to use the search features to find specific text within a file in the File Editor pane. See:

- Searching for text
- Searching and replacing text on page 13-4
- Searching for selected text on page 13-6.

13.2.1 Searching for text

To quickly find a text string in the current source file displayed in the File Editor pane:

1. Enter the required text string into the Find data field above the window.
2. Press the Enter key to find the first occurrence.
3. Press Enter again to find the next occurrence.

As you enter a search string, RealView Debugger copies the string to the clipboard. This means that the most-recently used string is available when you next display the Find, or Find and Replace, dialog box.

To find a text string in the current source file using the Find dialog box:

1. Position the cursor in the current file.
2. Select Find → Find... to display the Find dialog box, shown in Figure 13-1.

![Figure 13-1 Find dialog box](image)

3. Enter the required text string into the Find field in the dialog box. The last search string used might already be displayed as copied from the clipboard.
4. Click the Find button.
5. When a match is highlighted either:
   - click the Find button again to search for the next occurrence of the text string
Searching and replacing text

- replace the current selection, see Searching and replacing text.

If you have configured your search controls to enable wrapping, that is the default, and you do not replace all matched strings, the search cycles through the file and returns to the starting point. Then it begins again and continues until you click Close in the Find dialog box.

When the first match is found, you can move the focus to the File Editor pane and press F3 to find the next occurrence of the text string. It is not necessary to close the Find dialog box.

**Changing the search direction**

With the Find dialog box displayed, you can change the direction of the search regardless of the default, see Configuring searches on page 13-15. This might be useful if you miss a previous match and want to retrace your steps to verify the occurrence or to make a replacement.

**Searching multiple tabs**

If you display two or more file tabs in the current Code window then the search is restricted only to the file tab that is currently visible. However, you can select another file tab without closing the Find dialog box so continuing the search.

**Regular expressions**

With the Find dialog box displayed, you can select the Reg-expression check box so that it is checked. This forces the File Editor to treat the search string as a regular expression. See Using regular expressions on page 13-17 for more details on using this feature.

### 13.2.2 Searching and replacing text

To find and replace text in the current source file displayed in the File Editor pane:

1. Position the cursor in the current file.
2. Select **Find → Replace...** to display the Find and Replace dialog box. This has the same controls as the Find dialog shown in Figure 13-1 on page 13-3.
3. Enter the required text string into the Find field.
4. Click the **Find** button.
5. If there is no occurrence of the search string in the current file then an information box is shown saying that there is no match, the default, or the Code window flashes.

6. When a match is highlighted either:
   • click the Find button again to search for the next occurrence of the text string
   • replace the current selection.

7. Click Close to close the Find and Replace dialog box.

Making global changes

The Find and Replace dialog box also enables you to specify a global change where every occurrence of the search string is replaced in a single operation. Use the Replace All button to make a global change.

The search stops when all the matches have been replaced. There is no report detailing the number of replacements made but searching for the same string again shows that it no longer exists in the current file.

If you want to make the same global change in a second file tab then switch to the new tab without closing the Find and Replace dialog box and click Replace All again. In this way all occurrences of the text string are replaced in the second file.

Case sensitivity

By default, searches are case sensitive, that is, uppercase and lowercase are treated as distinct. Select the Inensitive Searches check box on the Find or the Find and Replace dialog box to change this for the current debugging session. When replacing a text string, the replacement string is always used exactly as entered in the Replace field. You must also enter spaces if required, in both the Find and Replace fields, because these are matched exactly.

Undoing replacements

You can reverse any search and replace operation. To do this, select the Edit → Undo option to reverse the last action. Where you have used the Replace All button, you must undo each replacement in turn as there is no Undo option for a global replacement.
13.2.3 Searching for selected text

You can search for text without invoking the Find dialog box:

1. Highlight the required search text in the current file tab.
2. Select Find → Find From Selection from the main menu.

This immediately causes the cursor to jump to the next occurrence of the selected text in the current file.

Select Find → Find Next from the main menu to look for the next occurrence. Or display the Find and Replace dialog box to make a replacement. The search stops when all the matches have been identified.

You can also use this method to search for selected text across multiple file tabs in the same File Editor pane.
13.3 Searching multiple files

Select Find → Find in Files... to display the Find in Files dialog box, shown in Figure 13-2. Use this to:

- search for a text string across multiple files
- search for a text string by specifying how it is used in the code.

Enter data into the following fields to specify the search:

**Search For:** Enter the search string. Later you can limit the search by specifying how this string is used in the source file.

**File Filter:** Use this field to specify the file types included in the search. You can limit the search to files having a particular extension or to files starting with a chosen letter. The file filter can also contain regular expressions built with the operators ?, *, [ ], and -not_match, for example [ms]*.c or [m?s?]*.[ch]. Click the File Filter button to display a list of previously used filters.

**Working Dir:**

This field contains the pathname of the default working directory which is the starting point for the search. You can click the Working Directory button to see a list of previously used pathnames to change this entry.
Search Type:

You can limit the search by specifying how the search string is used in the code. Select the usage you require from the Search Type list:

- **General String** searches for a text string regardless of how it is used.
- **Variable/Function Use** looks only for a string used as a variable or function parameter.
- **Function Definition** finds strings in a function definition but does not find calls to the function.
- **Macro Definition** finds strings in a macro definition but does not find calls to the macro.
- **Variable Definition** searches for strings in a variable definition but does not match uses of the variable.
- **Class Definition** searches for strings in a class definition but does not match uses of the class.
- **Pointer Variable Definition** finds strings in a pointer variable definition. It does not match uses of the pointer variable nor does it find places where the variable was defined but not as a pointer.
- **Typedef Definition** finds only strings that match the defined type.

Case Insensitive

Makes any searches case insensitive, that is treating uppercase and lowercase as identical. This is independent of the status of the check box on the Find, or the Find and Replace, dialog box.

Look in Subfolders

Includes subfolders when searching the working directory.

As you construct the search, the Find Command field shows the grep-style command that RealView Debugger will run. You can edit this command in this field. When the command is complete, click OK to start the search.

If you want to abandon the search, click the Cancel button and close the Find in Files dialog box, or click the Stop Build/Compile/Find button on the main toolbar.

13.3.1 Viewing search results

The results of any Find in Files search are displayed in two ways:
- a summary of matches is shown in the FileFind tab in the Output pane
- matching files can be displayed in the File Editor.
Viewing matches in the FileFind tab

Click the OK button in the Find in Files dialog box to make the FileFind tab current in the Output pane and display the grep -f command being executed. If the search finds matches then the results are displayed in the FileFind tab.

If the search finds matches then the results are displayed in the FileFind tab where the first matching occurrence of the search string is highlighted. Each entry shows:

- the filename in the chosen search path where a match has been found
- the line number in the file where the match has been found, set up by default using the -N flag
- the contents of the matching line.

Double-click on a chosen result line to move through the results list shown in the FileFind tab.

Viewing matches in the File Editor

If the current File Editor is empty, or you are editing a file not included in the results list, then a successful search creates a new file tab in the File Editor pane. This contains the first file in the search results and you can see a blue indicator pointing at the first matching line in the file. The matching line of code is also highlighted in the FileFind tab in the Output pane.

If you move down to a match in the results list and the corresponding file is not currently displayed in the File Editor then a new file tab is created automatically and the indicator positioned at the matching line.

If the File Editor already contains file tabs for all the files where successful matches have been identified then the corresponding tabs are displayed as you move through the results list in the Output pane.

The Find in Files... option searches for matches in files already stored on the disk. If you are editing a file in the File Editor that has not been saved to disk and a match is found in this file, then line numbers and the position of the indicator might not be shown accurately in the file tab. Therefore, to ensure that matches are located correctly, you must save any files before starting the search.
13.4 Working with functions

You can use the File Editor Find menu to search for, and move between, functions when editing your source files:

- Jumping to a function
- Jumping to a function definition
- Returning from a jump on page 13-11.

13.4.1 Jumping to a function

Select Find → Jump to Function... to display the Jump to Function dialog box to see a list of all the functions in the source file shown in the current file tab. If you select another file tab and then press Scan for Functions the list is refreshed for the new source file.

By default, the functions list is displayed sequentially, that is in the order the functions occur in the source file. Click the Sort check box to re-order the functions list into alphabetical order.

Click Jump, or double-click on the required name, to locate a function in your source file. This scrolls the file in the File Editor pane so that the first line of the function is visible towards the top of the pane. Click Close to close the Jump to Function dialog box and place the cursor at the start of the function ready for editing.

When editing a large file, the Jump to Function dialog is useful to find your way around. Select Find → Jump to Function... to display the Jump to Function dialog box containing the functions list for the current source file and the controls:

- **Show Curr**: Moves the functions list highlight bar to show which function currently contains the cursor.
- **Jump Curr**: Scrolls the file in the File Editor pane so that the first line of the function, currently containing the cursor, is towards the top of the pane. Click Close to close the Jump to Function dialog box and to place the cursor at the start of the function ready for editing.

13.4.2 Jumping to a function definition

If you are editing a large file in the File Editor, it is useful to be able to move to a function definition from the function call. In Example 13-1 on page 13-11 the cursor is located at the call to Func_1. Select Find → Jump to Function/Include at Cursor to move the cursor to the start of the definition of the function.
Example 13-1 Function call example

```c
Int_Loc = 2;
while (Int_Loc <= 2)
    if ( Func_1(Str_1_Par_Ref[Int_Loc],
                Str_2_Par_Ref[Int_Loc+1]) == Ident_1)
```

13.4.3 Returning from a jump

If you have used **Jump to Function...** or **Jump to Function/Include at Cursor** to jump to a function or function definition in the current source file, you can jump back to the starting point.

Select **Find → Jump Back** to reverse the last jump and return the cursor to the location before the jump was made. Unlike the **Edit → Undo** option, the maximum number of return jumps you can make is fixed at 16.
13.5 Pair matching

You can use the File Editor to search for, and move between, structures when editing your source files. Using this feature enables you to search your source code for:

- structure definitions consisting of a list of declarations enclosed in braces, that is curly brackets or {}
- function definitions containing a list of parameter declarations enclosed in parentheses, that is round brackets or ()
- array declarations enclosed in square brackets or []
- characters enclosed by double quotes, for example printf("\n");
- character constants enclosed by single quotes, for example if (c == '\n')
- comments
- conditions specified by if/else
- conditions evaluated during preprocessing specified by #if/#else/#endif.

This section contains:
- Searching pairs
- Matching pairs on page 13-13
- Matching nested pairs on page 13-14
- Enclosing on page 13-14.

13.5.1 Searching pairs

Select the option Find → Pair Matching... to display the Pair Matching dialog box, shown in Figure 13-3. Use this to search for pairs in the current file tab.

Specify the type of structure you are looking for in the Match field. Click on the drop-down arrow to display the pair selection list where you can highlight the required pair delimiters.

Figure 13-3 Pair Matching dialog box
Configuring the search

The current cursor position defines the starting point but you can specify how the search is carried out:

- select the **Forward** radio button to search from the starting point to the end of the file
- select the **Backward** radio button to search from the starting point to the beginning of the file
- select the **Enclosing** radio button to look for the pair currently enclosing the cursor, see Enclosing on page 13-14 for more information.

If you have enabled wrapping during search operations, this does not affect searching for pairs. When the end or beginning of the file is reached the search stops and reports no more matches using an information box (the default option).

13.5.2 Matching pairs

Select **Find → Pair Matching...** to display the Pair Matching dialog box shown in Figure 13-3 on page 13-12. This dialog box contains:

- **Highlight** Begins the search and scrolls the file to display the first match. The code within the matching pair is then highlighted in the File Editor pane. The search locates the first delimiter in the matching pair and places the cursor after it.
- **Jump Begin** Begins the search and scrolls the file to display the first match. The search locates the first delimiter in the matching pair and places the cursor after it.
- **Jump End** Begins the search and scrolls the file to display the first match. The search locates the second delimiter in the matching pair and places the cursor after it.

When a pair is matched in your source code, the File Editor displays the result by highlighting the pair or repositioning the cursor. When moving the cursor, the new location is displayed in the Cursor location field. You can make the cursor visible by making the Code window active.

Where you have more than one matching condition in your source file, you can continue searching through the file until you reach the end. This displays an information box or flashes the Code window to show that no more matches have been found.
13.5.3 Matching nested pairs

You can search for pairs in source files containing nested structures in the same way. However, the way pairs are matched depends on the:

- starting point
- search direction, that is whether you select Forward or Backward.

13.5.4 Enclosing

When configuring your search, select the Enclosing radio button, from the Pair Matching dialog box, to find the nearest pair delimiters enclosing the cursor.

If the cursor is not inside a pair of the selected type when the Enclosing radio button is selected, then the No match information box is displayed or the Code window flashes.
13.6 Configuring searches

When you are working in the File Editor, there are two ways to configure your searches:

- Using the Find dialog box
- Changing your workspace settings.

13.6.1 Using the Find dialog box

You can configure how to search files in the File Editor. Select **Find → Find...** from the main menu to display the Find dialog box shown in Figure 13-1 on page 13-3.

Table 13-1 describes the options available from the Find dialog box.

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Backwards</td>
<td>By default, a file is searched forwards, that is from the current cursor position to the end of the file. Set this option if you want to reverse the search direction.</td>
</tr>
<tr>
<td>Reg-expression</td>
<td>By default, the File Editor assumes that the string in the Find field is a text string. Setting this option forces the editor to interpret the string as a regular expression.</td>
</tr>
<tr>
<td>Stop at Top/Bottom</td>
<td>By default, any search operation wraps in the current file. That is the search begins at the current cursor position, goes to the end of the file and then restarts at the top of the file until it reaches the starting point. Set this option to disable wrapping, that is to force the search to stop when it reaches the top or the bottom of a file.</td>
</tr>
<tr>
<td>Insensitive Searches</td>
<td>By default, any search is case sensitive. That is uppercase and lowercase are distinct so searching for ARM does not find Arm or arm. Setting this option means that uppercase and lowercase are treated as identical.</td>
</tr>
</tbody>
</table>

13.6.2 Changing your workspace settings

You can change search defaults by changing the Search controls in your current workspace or in your global configuration file. Make changes here so that these settings are used the next time RealView Debugger starts up.

If you are working with a workspace, select **Tools → Workspace Options...** to make these changes. If you are working without a workspace, select **Tools → Options...** to make these changes to your global configuration file so that they are available in future debugging sessions.
Table 13-2 describes search settings that you can change.

**Table 13-2 Search settings in workspace**

<table>
<thead>
<tr>
<th>Object</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>The direction followed by a search. The default is to search forwards, that is from the top to the bottom of the file. To change this, set to backwards.</td>
</tr>
<tr>
<td>Wrap</td>
<td>The default is to wrap during a search, that is to search to the end of the file and then to start again at the top until the starting point is reached. To change this, set to stop.</td>
</tr>
<tr>
<td>Sensitive</td>
<td>This determines whether uppercase and lowercase are treated as identical in searches. The default is case sensitive. To change this, set to insens.</td>
</tr>
<tr>
<td>Regexp</td>
<td>When set to T, or True, full grep-style regular expressions are used in searches. The default is False, that is not enabled. When working in vi command mode, regular expressions are enabled by default.</td>
</tr>
<tr>
<td>Fail</td>
<td>When a search fails to find a match, the File Editor can display an information box or flash the File Editor pane. Set to dialog by default, you can change this to flash. When working in vi mode, failing to find a match displays a message in the vi command line.</td>
</tr>
</tbody>
</table>

For full details on changing your workspace settings to define search controls, see Chapter 10 *Configuring Workspace Settings*.
13.7 Using regular expressions

The File Editor provides regular expression matching that is similar to the UNIX grep command. A regular expression is a text string composed of characters, some of which have special meanings. Therefore, the regular expression string used in the search describes one or more possible literal strings. Similarly, you can use regular expressions to both search for and replace strings in your source files.

Using regular expressions enables you to carry out complex searches when editing your source files. This section gives a brief introduction. For a comprehensive reference guide to regular expressions, refer to *Mastering Regular Expressions* edited by Jeffrey E.F. Friedl.

This section contains:
- About special operators
- Searching with regular expressions on page 13-18.

13.7.1 About special operators

Table 13-3 shows the characters that have special meanings in a regular expression. In some cases, their meaning depends on where they occur in the regular expression.

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>The <em>match-any-character operator</em> matches any single character.</td>
</tr>
<tr>
<td>*</td>
<td>The <em>match-zero-or-more operator</em> repeats the smallest preceding regular expression as many times as necessary (including zero) to match the pattern.</td>
</tr>
<tr>
<td>+</td>
<td>The <em>match-one-or-more operator</em> repeats the preceding regular expression at least once, and then as many times as necessary to match the pattern.</td>
</tr>
<tr>
<td>?</td>
<td>The <em>match-zero-or-one operator</em> repeats the preceding regular expression once or not at all.</td>
</tr>
<tr>
<td>\n</td>
<td>The <em>back-reference operator</em> refers to a literal character within the regular expression.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
13.7.2 Searching with regular expressions

By default, text searches do not assume regular expressions. You can choose to set up your searches so that regular expressions are assumed, either:

- change the Search controls in your current workspace
- select the Reg-expression option in the Find (and Replace) dialog box.

Use the second method to configure regular expression matching for the current session only. When set, displaying the Find, and Find and Replace, dialog box opens with the radio button selected. You must change your workspace settings to enable this feature for all sessions.

To enable regular expression searches for the current File Editor session:

1. Select Find → Find... or select Find → Replace... to start a search sequence and replace matched text.
2. Select the Reg-expression option on the Find (or Find and Replace) dialog box.
3. Enter the required expression in the Find field.

The following examples show how to use regular expressions in search operations.
Matching simple expressions

Most characters in a regular expression match themselves. For example, entering the regular expression `struct` in the Find field matches all occurrences of the string `struct` in your source file. This changes, however, when the regular expression contains metacharacters as listed in Table 13-3 on page 13-17.

To match a metacharacter literally, precede the metacharacter with a backslash. For example, to find every occurrence of a dollar sign ($), type `\$` in the Find field. The backslash instructs the File Editor to interpret the dollar sign as a literal character, rather than a special character. If you do not use the backslash, the search finds the end-of-line characters instead.

Matching any single character

Using a dot or period (full stop) matches any single character so entering the regular expression `var.` in the Find field matches any four character sequence beginning with `var`, such as `var1`, and `var2`, and `var` followed by a space. It does not match `var` followed by an end-of-line character.

Matching alternative expressions

Using an alternation operation or pipe (|) matches alternative expressions so entering the regular expression `REG|Glob` in the Find field matches either `REG` or `Glob`. This can also be combined with the `not` operator, for example `/[^REG|Glob]/` which applies to both `REG` and `Glob`.

Matching repeating expressions

The following metacharacters enable you to match repeating occurrences of a regular expression in your search string:

- a regular expression followed by an asterisk, `*`, matches none, one, or more occurrences of that regular expression
- a regular expression followed by a plus sign, `+`, matches one or more occurrences of that regular expression
- a regular expression followed by a question mark, `?`, matches none or one occurrence of that regular expression.
Table 13-4 describes some simple examples to illustrate matching repeating expressions.

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>s*ion</td>
<td>None, one, or more occurrences of the s character immediately preceding the characters ion. This regular expression matches, for example, with ion in information, sections, and expressions, and with ssion and sion in expressions.</td>
</tr>
<tr>
<td>s+ion</td>
<td>One or more occurrences of the s character immediately preceding the characters ion. This regular expression matches, for example, the ssion and sion in expressions.</td>
</tr>
<tr>
<td>s?ion</td>
<td>None or one occurrence of the s character immediately preceding the characters ion. This regular expression matches, for example, with the sion in expressions, and with ion in information, sections, and expressions.</td>
</tr>
<tr>
<td>0.?</td>
<td>The number zero, followed by a dot or period (full stop). The backslash tells the File Editor to treat the dot as a literal character, and the ? operator acts on the dot. This regular expression matches, for example, with 0. and 0 followed by a character or an end-of-line character.</td>
</tr>
</tbody>
</table>

The asterisk, question mark, and plus metacharacters can operate on both single character regular expressions and grouped regular expressions. See *Grouping expressions* for details.

**Grouping expressions**

If an expression is enclosed in parentheses or round brackets ( ), it is treated as a single unit and repetition operators, such as the asterisk or plus sign are applied to the whole expression.

For example, to find strings that match is, you can type the text string is in the Find field. However, you can also use ( i)s as a regular expression. This instructs the File Editor to look for the letter s, preceded by both a space and the letter i. So, while using the text string search is matches with This, this, and is, the regular expression ( i)s matches only with is.
Matching any character in a list

A string of characters enclosed in square brackets ([ ]) matches any one character in that string. For example, the regular expression [xyz] matches any of the characters x, y, or z.

To match any character that is not in the string enclosed within the square brackets, precede the enclosed expression with a caret or not operator (^). For example, the regular expression [^abc] matches every character in the search text other than a, b, and c.

To specify a range of consecutive characters, use a minus sign (-) between the start and end characters, and place the whole expression within square brackets. For example, the regular expression [0-9] is the same as [0123456789].

The following applies to characters within the square brackets:

- If a minus sign is the first or last character within the square brackets, it is treated as a literal character. For example, the regular expression [-bc] matches any one of the characters -, b, and c.
- A right square bracket immediately following a left square bracket does not terminate the string. It is considered to be one of the characters to match. For example, the regular expression []0-9 matches the right square bracket and any digit.
- Metacharacters, such as backslash \, asterisk *, or plus sign +, immediately following the opening square bracket are treated as literal characters. For example, the regular expression [. ] matches the dot or period (full stop).

You can use square brackets to group regular expressions in the same way as parentheses. The text string in the square brackets is treated as a single regular expression. For example, the regular expression [bsl]ag matches any of bag, sag, or lag while [aeiou][0-9] matches any lowercase vowel followed by a number, such as a1.

Matching the start or end of a line

You can use a regular expression to search for start-of-line and end-of-line characters:

- If a caret, ^, is at the beginning of the entire regular expression, it matches the beginning of a line. For example, the regular expression ^reg_opt matches any occurrence of reg_opt but only at the start of a line.
- If a dollar sign, $, is at the end of the entire regular expression, it matches the end of a line. For example, reg_opt$ matches any occurrence of the string reg_opt but only at the end of a line.
• If an entire regular expression is enclosed by a caret and dollar sign (^par_a4 == reg_opt$), it matches an entire line.

You can build complex search strings by combining regular expressions and metacharacters, for example ^(aeiou)[0-9] or ([aeiou][0-9]$).
Chapter 14

Working with Version Control Systems

This chapter describes RealView Debugger support for version control software to manage access to source files and other files associated with debugging your applications, such as user-defined projects. It contains the following sections:

- Defining the version control tool on page 14-2
- Using a version control tool on page 14-3
- Configuring a custom version control tool on page 14-8.
14.1 Defining the version control tool

When you first run RealView Debugger after installation (or without a defaults file), it attempts to identify your version control tool:

1. Is ClearCase specified in the Registry?
2. Is PVCS specified in the Registry and the PCVS executable on the path?
3. Is the environment variable CLEARCASE_ROOT set?
4. Is the environment variable CVSROOT set?

If all checks fail then RealView Debugger assumes that you do not have a version control system.

If RealView Debugger fails to identify your version control tool, you must specify it. See Configuring a custom version control tool on page 14-8 for details on how to do this.

——— Note ————

RealView Debugger continues to use the specified version control tool until you change it explicitly.

14.1.1 Version control under UNIX

If you are using UNIX, the environment variables CLEARCASE_ROOT and CVSROOT are examined to identify the version control tool.
14.2 Using a version control tool

In this section, RealView Debugger is running on a Windows workstation using WinCVS to manage source files. It describes:

- Changing the status of controlled files
- Using source control commands
- Setting up a prompt on page 14-7.

14.2.1 Changing the status of controlled files

Opening a file that is under version control, activates the Source control button on the Editing toolbar. This button shows the read/write status of the current file:

- **Locked**: Indicates that the file is locked. A file marked in this way cannot be edited without changing the status.
- **Read-Write**: Indicates that the file is editable. A file marked in this way can be edited without changing the status.
- **Read-only**: Indicates that the file is read-only. A file marked in this way cannot be edited without changing the status.

If a file is read-only in the File Editor pane, and you try to edit, RealView Debugger displays a prompt, where you can choose to change the file status so that editing can take place.

Click **Yes** to change the file status so that it can be edited. This changes the icon displayed on the Source control button. You can also change the status of the file by clicking on the Source control drop-down arrow to display the Source control menu.

Source control buttons, and associated menu options, are also available from standalone editors, and are used in the same way.

14.2.2 Using source control commands

Click the Source control button drop-down arrow to access the source control commands. The options available from this menu depend on the status of the file loaded into the File Editor pane and the version control tool you are using.

**Locked, read-only files**

If you are working with a locked file, click the Source control button drop-down arrow to display the Source control menu shown in Figure 14-1 on page 14-4.
The menu options are:

**<Not Editable>**

Indicates the status of the file as Locked and so it cannot be edited.

**Raise WinCVS**

Starts up the version control tool.

**Prompt before Submitting**

Source control commands chosen from the Source control menu are submitted directly for execution. Click this option to display a prompt showing the command before submission, see *Setting up a prompt* on page 14-7 for details.

If you are working with a read-only file, the Source control menu includes:

**Allow Editing**

Click to change the status of the file to Read-Write so that it can be edited. RealView Debugger warns if the contents of the buffer have changed since the file was last saved.

**Read-Write files**

If you are working with an editable file, click the Source control button drop-down arrow to display the Source control menu shown in Figure 14-2 on page 14-5.
In this example, the menu options include:

**Make Buffer Read-Only**
Changes the status of the file to Read-only so that it cannot be edited.

**Check-In File**
Checks the current file into the repository. Selecting this option displays a message dialog box where you can enter a text comment, for example *Updated for new tools env*. This is used as the message to identify the check-in, for example:

```
cvs commit -m "Updated for new tools env" filename
```

**UnCheck-Out File**
Updates the current file to synchronize it with any changes saved to the repository since it was last checked out or committed, for example:

```
cvs update filename
```

The original file is renamed, for example *dhry_1.c.old*.

**Enter File into Source Control**
Adds the current file to the repository, for example:

```
cvs add filename
```

**Raise WinCVS**
Starts up the version control tool.

**Show Files You have Checked Out**
Updates the current file to merge changes. Where a file contains changes, this is roughly equivalent to a checkout command, for example:

```
cvs -nq update
```
This does not complete the update, defined by the -n flag. It is also quiet (-q flag).

**Show Version History**

Displays the history information for the current file, for example:

```bash
cvs -q log -N filename
```

This does not list tags, defined by the -n flag. It is also quiet (-q flag).

**Show Changes from Previous Version**

Displays differences between the current file and the previous version in the repository, for example:

```bash
cvs diff filename
```

**Show Differences of two Versions**

Displays differences between two versions of the file currently checked out. Selecting this option displays the first of two prompts where you specify the version numbers to compare, for example:

```bash
cvs diff -r 1.1 -r 1.3 filename
```

**Show Tags**

Displays status information for the current file, including tags, for example:

```bash
cvs status -v filename
```

**File Status**

Displays status information for the current file, for example:

```bash
cvs status filename
```

**Merge Changes**

Updates the current file to merge changes. Where a file contains changes this is equivalent to a checkout command, for example:

```bash
cvs update filename
```

This does complete the update.

If you are working with a custom version control tool, that is one that is not supported by RealView Debugger by default, you can specify the commands submitted. These appear as options on the menus described in this section. See *Specifying custom commands* on page 14-10 for details on setting up these custom commands.
14.2.3 Setting up a prompt

By default, source control commands are submitted directly for execution. This is defined in your workspace settings file. You can configure this:

- change the `Src_ctrl` settings in your workspace, described in Configuring a custom version control tool on page 14-8
- set up a temporary prompt.

To set up a temporary prompt, select **Prompt before Submitting** from the **Source control** menu, shown in Figure 14-2 on page 14-5. From this point on, RealView Debugger displays the prompt, shown in Figure 14-3, to confirm a command submission.

![Figure 14-3 Source control command prompt](image)

The data field shows the command ready for submission. You can edit the command in this field before submission. Click **Submit** to execute the command shown in the command field.

Click **Cancel** to abort the submission and close the prompt box.

If you submit version control commands, the **SrcCtrl** tab is brought to the front of the Output pane. This displays submitted commands and any messages returned from your version control tool.
14.3 Configuring a custom version control tool

You can specify the version control system you are using or set up RealView Debugger to handle a custom tool. If you choose to specify a custom tool, you must also define the commands to use.

To configure version control settings you must change your workspace settings. This means that the settings are re-used when RealView Debugger next starts up with the current workspace. This section describes how to do this:

- Changing your workspace settings
- Specifying custom commands on page 14-10.

Note

If you are working without a workspace, select Tools → Options... to display the Options window to make the changes described in the rest of this chapter. Changing version control rules in your global configuration file means that they apply across all workspaces and are available when you are working without a workspace.

14.3.1 Changing your workspace settings

To examine, or change, your current workspace settings, select Tools → Workspace Options.... This displays the Workspace Options window, described in detail in Chapter 10 Configuring Workspace Settings.

To configure version control settings for the current workspace:

1. Expand the hierarchy to display the Src_ctrl settings in the rvdebug.ini file, shown in Figure 14-4 on page 14-9.
2. Change the values as required, for example select the toggle box to set Query to True. Changing this setting displays the command prompt for confirmation before submitting a command.

3. Select File → Save and Close to save the new settings and close the Workspace Options window.

Table 14-1 gives a summary of source control settings that can be configured in the workspace settings file.

Table 14-1 Src_ctrl settings in workspace

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Type of version control system. Right-click to see a list of suggested entries.</td>
</tr>
<tr>
<td>Name</td>
<td>Name of version control tool.</td>
</tr>
<tr>
<td>Cust1</td>
<td>The menu name for the first custom command. For example, this replaces the option Show Tags in the Source control menu, shown in Figure 14-2 on page 14-5.</td>
</tr>
<tr>
<td>Cust2</td>
<td>The menu name for the second custom command. For example, this replaces the option File Status in the Source control menu, shown in Figure 14-2 on page 14-5.</td>
</tr>
<tr>
<td>Query</td>
<td>Specifies if a command prompt is displayed for confirmation before submission. This is set to False by default.</td>
</tr>
</tbody>
</table>
14.3.2 Specifying custom commands

If you specify a custom control tool, you must also specify the valid source control commands:

1. Expand the hierarchy to display the source control settings Cmds container in the rvdebug.ini file, shown in Figure 14-5.

2. Change the values as required, for example right-click on the Co string entry and select Edit Value from the context menu. Enter the command used to check files out for editing as defined by your custom control tool.

3. Select File → Save and Close to save the new settings and close the Workspace Options window.

--- Note ---

You can also use these settings to override known commands, for example to replace a command with an alias.
Table 14-2 gives a summary of custom command settings that can be configured in the workspace settings file.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Co</td>
<td>Command to check files out for editing</td>
</tr>
<tr>
<td>Ci</td>
<td>Command to check files in for editing</td>
</tr>
<tr>
<td>Unco</td>
<td>Command to cancel checkout</td>
</tr>
<tr>
<td>Add</td>
<td>Command to add a file to source control</td>
</tr>
<tr>
<td>Tool</td>
<td>Source control tool to use</td>
</tr>
<tr>
<td>Colist</td>
<td>Command to show checkout list</td>
</tr>
<tr>
<td>Vershist</td>
<td>Command to show version history</td>
</tr>
<tr>
<td>Diff</td>
<td>Command to show differences between two versions</td>
</tr>
<tr>
<td>Diffprev</td>
<td>Command to show differences between current and previous versions</td>
</tr>
<tr>
<td>Cust1</td>
<td>First custom command, submitted when the first custom command is selected from the <strong>Source Control</strong> menu</td>
</tr>
<tr>
<td>Cust2</td>
<td>Second custom command, submitted when the second custom command is selected from the <strong>Source Control</strong> menu</td>
</tr>
</tbody>
</table>

**Note**
Changes made to these workspace settings take immediate effect. However, setting custom menu names to empty does not take effect until RealView Debugger is restarted because this is the point when defaults are restored.
Appendix A

Workspace Settings Reference

This appendix contains reference details about settings that define the workspace and the global configuration options. It contains the sections:

- `DEBUGGER` on page A-2
- `CODE` on page A-6
A.1 DEBUGGER

Settings in this group govern the behavior of generic actions in the debugger. These controls are then used in conjunction with other processor-specific controls.

DEBUGGER contains two second-level groups and a file:
- Command
- Disassembler on page A-3
- Board_file on page A-4.

A.1.1 Command

Settings in this group control the behavior and appearance of the Code window command line and Output pane. Use these to customize the input and output format used in this area.

When RealView Debugger starts, it uses the last-used settings unless overridden by settings in this group. These settings can be overridden dynamically by issuing CLI commands.

Saving changes takes immediate effect or at next start-up.

Table A-1 describes the settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num_lines</td>
<td>The height of the Output pane. The default setting is 5 lines. Values can</td>
</tr>
<tr>
<td></td>
<td>be entered in hex or decimal, for example 15 or 0x000F.</td>
</tr>
<tr>
<td>Radix_in</td>
<td>This setting specifies the format of number input options at start-up. The</td>
</tr>
<tr>
<td></td>
<td>default format is decimal which allows hex numbers to be entered, for</td>
</tr>
<tr>
<td></td>
<td>example 0xABCD and 0'ABh. Switching to hex allows decimal numbers to be</td>
</tr>
<tr>
<td></td>
<td>entered, for example 01234t.</td>
</tr>
<tr>
<td>Radix_out</td>
<td>This setting specifies number format output options at start-up. The</td>
</tr>
<tr>
<td></td>
<td>default format is decimal.</td>
</tr>
</tbody>
</table>
A.1.2 Disassembler

Settings in this group control how the disassembly view is displayed in the Code window. This can be set for all processors or for specific processors only. The default settings apply to all processors.

When RealView Debugger starts, it uses the last-used settings unless overridden by these settings. These settings can be overridden dynamically by issuing CLI commands.

Saving changes takes immediate effect.

--- Note ---
Some processor disassemblers do not support features configured with these settings, and the settings are ignored.

Table A-2 describes the settings.

Table A-2 Disassembler settings

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbols</td>
<td>When instructions reference direct memory locations, either relative to the PC or as absolute references, the debugger tries to show the symbol at that location. Use this to disable this property.</td>
</tr>
<tr>
<td>Labels</td>
<td>When an instruction has a label associated with the address, the debugger shows it inline. Use this to disable this property.</td>
</tr>
<tr>
<td>Source</td>
<td>By default, high-level source code is interleaved with disassembly code when available. Use this to disable this property.</td>
</tr>
</tbody>
</table>
A.1.3 Board_file

Change this setting to specify a different board file for the current session.

Changing this value takes immediate effect. The specified board file is read and the contents used to populate the configuration details.
Resetting the value back to Empty does not take effect until the next time RealView Debugger starts.
A.2 CODE

Settings in this group govern the behavior of all Code windows when running a
debugging session. These settings control the display characteristics of windows, their
size and position, and any user-defined buttons created on the toolbars (not available in
this release).

CODE contains two second-level groups and a settings rule:
• Pos_size
• Button
• Asm_type on page A-7.

A.2.1 Pos_size

Settings in this group control the position on screen and the size of Code windows in
lines and characters. Use these to customize the size and position of Code windows in
the debugger.

On start-up, RealView Debugger uses the last-used settings unless overridden by these
settings. As you open new Code windows in the session, they are controlled by these
settings.

Table A-3 describes the settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num_lines</td>
<td>Use this to specify window height as a given number of lines. The default is 0x0200.</td>
</tr>
<tr>
<td>Num_chars</td>
<td>Use this to specify window width as a given number of characters. The default is 0x02A0.</td>
</tr>
<tr>
<td>X_pos</td>
<td>Use this to specify the X position of the top-left corner of the window. The default is 0x010F.</td>
</tr>
<tr>
<td>Y_pos</td>
<td>Use this to specify the Y position of the top-left corner of the window. The default is 0x0066.</td>
</tr>
</tbody>
</table>

A.2.2 Button

Use these settings to create new buttons to customize Code window toolbars. You can
also use these settings to edit default buttons or to replace Default with a new group that
defines a button of your design.
When it first starts, RealView Debugger uses the last-used settings unless overridden by these settings. Any new Code windows you then open in the session are controlled by these settings.

Table A-4 describes these settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hover</td>
<td>Use this to specify the hover text to accompany the new or edited button. By default, the button name is used as the hover text unless overridden by this setting.</td>
</tr>
<tr>
<td>Description</td>
<td>Use this to specify the description text that is shown in the Status line when the pointer is over the new or edited button. You can also specify a file to contain the text.</td>
</tr>
<tr>
<td>Send_to</td>
<td>When creating a new button, you must specify a destination, for example a debugger command, a registered DLL, or the operating system command shell.</td>
</tr>
<tr>
<td>Command</td>
<td>Use this to specify the command generated by the new button.</td>
</tr>
<tr>
<td>Icon</td>
<td>Use this to specify the icon displayed for the new button. You can select from the list of predefined icons. Right-click to see the available options.</td>
</tr>
<tr>
<td>Dll</td>
<td>Use this to specify the DLL underneath the new button, if any.</td>
</tr>
<tr>
<td>Position</td>
<td>Use this to specify the location of the new button on the toolbar. You can place the button relative to existing buttons or append it.</td>
</tr>
</tbody>
</table>

**Note**

These settings are not available in this release.

**A.2.3 Asm_type**

When an assembler source file opens, RealView Debugger decides what type of processor is in use. However, the processor type is unknown if:

- there are no active connections
- there are no user-defined projects currently open
- there are no auto-projects currently open.

In this case, RealView Debugger does not know the format of instructions and so cannot define source coloring rules. This generates a selection box where you can specify the processor type.
Change this to specify a default processor type on start-up, for example ARM.

Saving a change to this setting takes immediate effect on new assembler source files opened following the update.
A.3 ALL

This group contains three second-level groups:

- Text
- Search on page A-10
- Edit on page A-11.

A.3.1 Text

Settings in this group control the File Editor pane and editor functions within the Code window.

Saving changes might not take effect until the next time RealView Debugger starts, or when a new Code window opens, or when a standalone editor starts.

The Text group contains one third-level group and a series of settings:

Source_coloring

These settings control the colors used to identify source tokens. The defaults have been chosen to be easy to read and work well to isolate different program areas. The coloring choices are made relative to the built-in color models.

Table A-5 describes the settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>File_extensions</td>
<td>The standard C/C++ source coloring is auto-enabled based on file extension. Use this to specify a comma-separated list of file extensions that, when loaded, trigger source code coloring.</td>
</tr>
<tr>
<td>Numbers</td>
<td>Use this to specify the color for numbers displayed in the File Editor pane or in a standalone editor window.</td>
</tr>
<tr>
<td>Strings</td>
<td>Use this to specify the color for strings displayed in the File Editor pane or in a standalone editor window.</td>
</tr>
<tr>
<td>Keywords</td>
<td>Use this to specify the color for C/C++ keywords displayed in the File Editor pane or in a standalone editor window.</td>
</tr>
<tr>
<td>Comments</td>
<td>Use this to specify the color for comments displayed in the File Editor pane or in a standalone editor window.</td>
</tr>
</tbody>
</table>
Settings in this group configure editor behavior when working with source files in the File Editor pane or in a standalone editor window. Table A-6 describes these settings.

### Table A-6 Text settings

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>Use this to specify the height, in number of lines, for text displayed in the File Editor pane or in a standalone editor window.</td>
</tr>
<tr>
<td>Width</td>
<td>Use this to specify the width, in number of characters, for text displayed in the File Editor pane or in a standalone editor window.</td>
</tr>
<tr>
<td>Src_color_dis</td>
<td>Source coloring is used to make it easier to read source of high-level and low-level languages. All source coloring can be disabled in which case all text will be the same color (usually black). By default, source coloring is enabled, that is this setting is False.</td>
</tr>
</tbody>
</table>

#### A.3.2 Search

Settings in this group control the searching behavior when working with source files in the File Editor pane.

These settings can be overridden dynamically using the menus and toggles in the File Editor.
Table A-7 describes these settings.

Table A-7 Search settings

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Use this to specify the search direction. The default is to search forwards, that is, from the top to the bottom of the file.</td>
</tr>
<tr>
<td>Wrap</td>
<td>Use this to specify search behavior when the end of file is reached. The default is to wrap during a search, that is, to search to the end of the file and then to start again at the top until the starting point is reached.</td>
</tr>
<tr>
<td>Sensitive</td>
<td>Use this to specify whether uppercase and lowercase characters are treated as identical in searches. By default, searches are case-sensitive.</td>
</tr>
<tr>
<td>Regexp</td>
<td>When set to True, full grep-style regular expressions are used in searches. The default is False, not enabled. When working in vi command mode, regular expressions are enabled by default, unless disabled using the :set command.</td>
</tr>
<tr>
<td>Fail</td>
<td>Use this to specify editor behavior when a search fails. Set to dialog by default, you can change this to flash. When working in vi mode, a message is displayed in the vi command line when no match is found.</td>
</tr>
</tbody>
</table>

A.3.3 Edit

Settings in this group control general editor behavior when working with source files in the File Editor pane. These settings can also be used to control the operation of a standalone editor if specified for use outside RealView Debugger.

These settings can be overridden dynamically using the menus and toggles in the File Editor.

The Edit group contains three third-level groups and a series of settings:

Backup

These settings control the backup behavior when working with source files in the File Editor pane.
Table A-8 describes these settings.

**Table A-8 Backup settings**

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable</td>
<td>By default, a backup file is created when a file is edited. This provides a</td>
</tr>
<tr>
<td></td>
<td>useful safety feature. Use this to disable this feature if required.</td>
</tr>
<tr>
<td>Backup_dir</td>
<td>By default, backup files are saved in the same directory as the original</td>
</tr>
<tr>
<td></td>
<td>file. Use this to specify a pathname to a new location, for example to</td>
</tr>
<tr>
<td></td>
<td>keep all backup files in one special directory.</td>
</tr>
<tr>
<td>Backup_ext</td>
<td>By default, backup files are saved with the .bak extension appended to the</td>
</tr>
<tr>
<td></td>
<td>original filename.</td>
</tr>
</tbody>
</table>

**Tab_conv**

Settings in this group control the display behavior when working with source files in the File Editor pane. These settings are used to handle tabs and spaces.

Tabs are allowed in files and are left untouched, by default. Use these settings to convert tabs to spaces when writing to the file, that is saving, and to convert spaces to tabs when reading the file.

Spaces are not converted to tabs inside " and “ quoting blocks on a line.

Table A-9 describes these settings.

**Table A-9 Tab_conv settings**

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tabs_to_spaces</td>
<td>Converts tabs to spaces when the file is saved.</td>
</tr>
<tr>
<td>Spaces_to_tabs</td>
<td>Converts spaces to tabs when the file is read.</td>
</tr>
<tr>
<td>To_spaces_ext</td>
<td>Use this to specify file extensions where tab conversions take place.</td>
</tr>
<tr>
<td></td>
<td>Specify a list separated by semi-colons (;)</td>
</tr>
<tr>
<td>To_tabs_ext</td>
<td>Use this to specify file extensions where space conversions take place.</td>
</tr>
<tr>
<td></td>
<td>Specify a list separated by semi-colons (;).</td>
</tr>
</tbody>
</table>

**Src_ctrl**

Settings in this group control source control access tools when working in the File Editor pane and in the debugger. The editor attempts to detect any source control system in use, where possible, but you might have to specify it before RealView Debugger can use it.
Use these settings to specify complete source control commands, and to override commands for known systems.

The **Src_ctrl** group contains a low-level group and a series of settings:

**Cmds**
Use these to specify source control commands for use with RealView Debugger.

**Src_ctrl settings**
Use these to specify the source control system to be used with RealView Debugger to control access to files.

**Edit settings**
Settings in this group configure editor behavior when working with source files in the File Editor pane.

Table A-10 describes these settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drag_drop_dis</td>
<td>Use this to disable drag-and-drop editing when working in the File Editor pane.</td>
</tr>
<tr>
<td>Vi</td>
<td>Running the editor in vi mode enables you to access all the vi commands and most ex commands. You can configure the debugger to start the File Editor in vi mode using this setting. When in insert mode, all <strong>Common User Access</strong> (CUA) editing features are available. You can also enable this option from the File Editor pane menu bar.</td>
</tr>
<tr>
<td>Indent</td>
<td>Use this to set indenting so that a specified number of spaces are inserted as you open a new line. By default, auto-indent inserts the same number of spaces as on the previous line. If the previous line is a left curly bracket (<code>{</code>) the shift is increased. If the previous line is a right curly bracket (<code>}</code>), shift spaces are subtracted.</td>
</tr>
<tr>
<td>Undo</td>
<td>Use this to specify the levels of undo and redo. By default, this is set to 64.</td>
</tr>
<tr>
<td>Tab</td>
<td>Use this to specify the size of TAB settings when working in the File Editor. By default, this is set to 8. Use a value between 1 and 16.</td>
</tr>
<tr>
<td>Shift</td>
<td>Use this to specify the size of shift spaces as used in the Indent rule and accessed through the File Editor menu options. By default, this is set to 2. Use a value between 2 and 32.</td>
</tr>
<tr>
<td>Line_number</td>
<td>By default, line numbering is disabled in the debugger and the File Editor. Use this to change the editor default to show line numbers at start-up.</td>
</tr>
</tbody>
</table>
By default, tooltip evaluation of variables and registers is enabled. Change this setting to `True` to disable this feature.

**Timer**
During file editing, the editor periodically checks to see if another tool has edited or deleted the files being tested. A warning is shown if an update is detected. Use this to specify the number of seconds between checks. The default is 60 seconds. Use values greater than 30 seconds. Set to `-1` to disable this feature.

**Tool_save**
When performing a build, you are prompted to resave any files that have been edited. Use this to specify automatic resaving of changed files at build time to ensure that your latest sources are included. You can also set a no-save, no-ask value.

**Startup**
The default start-up file, that is `rvdebug.sav` in your home directory, contains a list of previously edited files and information from previous debugging or editing sessions. This enables historical information to be separated from your current session. Use this to specify a different start-up file, in a new location. Set this to `- (dash)` to specify that no start-up file is used.

**Template**
During file editing, you can use templates to speed up code development. The template file contains templates that you can use or edit as required. By default, the file is named `rvdebug.tpl` and is saved in your home directory, or in the default settings directory `\etc`. Use this to change this pathname.

**Restore_state**
This setting applies only to a standalone editor. Use this to start the standalone editor in the same state it was in when you last exited.

<table>
<thead>
<tr>
<th>Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>No_tooltip</td>
<td>By default, tooltip evaluation of variables and registers is enabled. Change this setting to <code>True</code> to disable this feature.</td>
</tr>
<tr>
<td>Timer</td>
<td>During file editing, the editor periodically checks to see if another tool has edited or deleted the files being tested. A warning is shown if an update is detected. Use this to specify the number of seconds between checks. The default is 60 seconds. Use values greater than 30 seconds. Set to <code>-1</code> to disable this feature.</td>
</tr>
<tr>
<td>Tool_save</td>
<td>When performing a build, you are prompted to resave any files that have been edited. Use this to specify automatic resaving of changed files at build time to ensure that your latest sources are included. You can also set a no-save, no-ask value.</td>
</tr>
<tr>
<td>Startup</td>
<td>The default start-up file, that is <code>rvdebug.sav</code> in your home directory, contains a list of previously edited files and information from previous debugging or editing sessions. This enables historical information to be separated from your current session. Use this to specify a different start-up file, in a new location. Set this to <code>- (dash)</code> to specify that no start-up file is used.</td>
</tr>
<tr>
<td>Template</td>
<td>During file editing, you can use templates to speed up code development. The template file contains templates that you can use or edit as required. By default, the file is named <code>rvdebug.tpl</code> and is saved in your home directory, or in the default settings directory <code>\etc</code>. Use this to change this pathname.</td>
</tr>
<tr>
<td>Restore_state</td>
<td>This setting applies only to a standalone editor. Use this to start the standalone editor in the same state it was in when you last exited.</td>
</tr>
</tbody>
</table>
Appendix B
Project Properties Reference

This appendix gives reference information on the Project Properties configuration options that are displayed in the Project Properties window when you select Project → Project Properties in a RealView Debugger Code window. See Chapter 11 Managing Projects for more information on using projects and configuring project settings.

This appendix contains the following sections:
- PROJECT group on page B-3
- PROJECT group for a Container project on page B-8
- SETTINGS group on page B-9
- CONFIGURATION group on page B-22
- COMPILE group on page B-24
- ASSEMBLE group on page B-44
- CUSTOM group on page B-56
- BUILD group on page B-58
- BUILD_LIB group on page B-69
- MAKEFILE group on page B-77.
Note

Many of the project properties described in this appendix are specific to ARM tools. If you are not using one of these, see your tool-specific documentation for a complete list of the project properties that are available.
B.1 PROJECT group

The PROJECT group contains settings that describe the basic project information. It is used to define project control. Most of the fields in this group are predefined and cannot be modified.

--- Note ---

If you create a user-defined project, you can merge the settings from an auto-project. However, the PROJECT group settings are not merged. Any changes you make to this group for an auto-project are lost if you merge settings. For more details, see Merging auto-project settings into a project on page 11-36.

Right-click on the PROJECT group and select Expand whole Tree, from the context menu, or double-click, to view the contents (see Figure B-1).

![Figure B-1 PROJECT group](image)

The PROJECT group contains:

- **Top-level PROJECT group settings** on page B-4
- **Command_Open_Close group** on page B-6
- **Modification_History group** on page B-7.
### B.1.1 Top-level PROJECT group settings

Table B-1 describes the settings available at the top level of the PROJECT group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specific_device</td>
<td>Specifies a named processor to bind the project to, enabling specific project binding or autobinding. If set, the project only binds to the named device. If this is not set, the project binds to all processors in the family specified by the Processor setting. Names given for this setting must match device names given in the corresponding JTAG file.</td>
</tr>
<tr>
<td>Description</td>
<td>Is a description of the project. You can also point to a file, for example a ReadMe file, where information is stored.</td>
</tr>
<tr>
<td>Processor</td>
<td>Specifies the toolchain that the project builds for. This entry cannot be modified.</td>
</tr>
<tr>
<td>Type</td>
<td>Defined when the project is created. It specifies the project type, for example Simple, Makefile, or Container. This entry cannot be modified.</td>
</tr>
<tr>
<td>Author</td>
<td>Defined when the project is created. It is derived from the user ID. This enables the project to be locked by the author. This entry cannot be modified.</td>
</tr>
<tr>
<td>Lock</td>
<td>Locking controls access to the project and the sources. Right-click to see the available options:</td>
</tr>
<tr>
<td></td>
<td>Unlocked The project files are not locked.</td>
</tr>
<tr>
<td></td>
<td>Versioned A version lock means that the source control locks are also imposed on the project. The project is checked into the source control system. If you attempt to change the project, it is checked out for modification. Checkin enables file stamping.</td>
</tr>
<tr>
<td></td>
<td>User A user lock means that only the author can modify the project</td>
</tr>
<tr>
<td>Base_directory</td>
<td>A base directory pathname enables all sources and build files to be relative to the given directory, if within it or in a subdirectory. Specifying a base directory also enables the source or build tree to be located in a different location than the project file. If not set, the directory of the project settings file is used as the default base directory for the project.</td>
</tr>
</tbody>
</table>
### Table B-1 PROJECT group (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Tool_directory** | Specifies the path to the build tools to use with the project.  
If not set, the project toolchain is used.  
If set, the directory given here is added to your path environment variable before running any tools. |
| **Tool_envvar** | Specifies any environment variables required for the build tools identified in the **Tool_directory** entry. These are set prior to running any tools. |
| **Source_search** | Specifies the source search paths where these are not supplied by the compiler or assembler for use by RealView Debugger.  
Each search path you define is identified by a **Source_search** entry.  
This is displayed in the Source Search Paths dialog box and overrides any paths set up by the `RVDEBUG` environment variable.  
If multiple pathnames are defined, select the **Manage List...** option from the context menu to control how the different paths are accessed. |
B.1.2 Command_Open_Close group

The Command_Open_Close group contains commands submitted to RealView Debugger when a project binds to or unbinds from a connection. For more information on the RealView Debugger commands, see RealView Debugger v1.6 Command Line Reference Guide.

Table B-2 describes the settings available in the Command_Open_Close group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open_conn</td>
<td>Specifies one or more commands to execute when the project binds to a connection. You can specify any command accepted by RealView Debugger, such as loading an INCLUDE file, reloading an image, or setting up breakpoints or macros. Open_conn is the setting for Standard and Library projects.</td>
</tr>
<tr>
<td>Open</td>
<td>Open is the setting for Custom and auto-projects.</td>
</tr>
<tr>
<td>Close</td>
<td>Specifies one or more commands to execute when the project unbinds from a connection. You can specify any command accepted by RealView Debugger, such as deleting macros created for the project.</td>
</tr>
</tbody>
</table>
B.1.3 Modification_History group

The Modification_History group contains a default, third-level group named Create, that gives details of the project creation. This group is created by RealView Debugger when the project is created.

You can create a new group, for example MY_UPDATE, to store details of changes that you make during the project lifecycle by copying the default creation group.

Table B-3 describes the settings available in the Create group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>The name of the user who originally created the project</td>
</tr>
<tr>
<td></td>
<td>This entry cannot be modified</td>
</tr>
<tr>
<td>Date</td>
<td>The date the project was created</td>
</tr>
<tr>
<td></td>
<td>This entry cannot be modified</td>
</tr>
<tr>
<td>Type</td>
<td>The type of action</td>
</tr>
<tr>
<td>Description</td>
<td>A short description of the action performed on the project</td>
</tr>
</tbody>
</table>
B.2 PROJECT group for a Container project

A Container project is defined using a small subset of the settings values described in this chapter. The PROJECT group includes a list of subprojects loaded by the project container. The order of the list defines the order in which RealView Debugger opens the projects (that is, the build order). Therefore, you must specify the Sub project list in the order of dependency.

You can include projects in the subprojects list that are disabled and so are not loaded by the Container project. These projects have an exclamation mark (!) appended at the start of the path name, shown in the example in Figure B-2.

![Figure B-2 PROJECT group settings values for a Container project](image)

Right-click on a Sub project setting in the Settings Values pane and select Enable Sub-Project from the context menu to re-enable a subproject.

——— Note ————

Container projects can be nested but not recursive, that is a container project can include other container projects but must not include itself. However, the nested container project has no makefile and so any build fails.
B.3 SETTINGS group

The SETTINGS group contains settings that provide additional control over your application during a debugging session, such as:

- image load commands
- predefined breakpoints.

Most of the fields in this group are set by default, but they can be modified as required. The SETTINGS group is described in:

- Top-level SETTINGS group values on page B-10
- Image_load group on page B-12
- Auto_Set_Breaks group on page B-14
- Named_Breaks group on page B-16
- Runtime_Control group on page B-18.

Note

If you have an auto-project, you can create a user-defined project for the image. If you have made any changes to the SETTINGS group for the auto-project, you can choose to merge these settings when you create the user-defined project. For more details, see Merging auto-project settings into a project on page 11-36.
B.3.1 Top-level SETTINGS group values

The SETTINGS group, shown in Figure B-3, contains settings that affect the loading of the image for a project, and for breakpoint control and runtime use.

![Figure B-3 SETTINGS group](image)

Table B-4 describes the settings available in the SETTINGS group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open_load</td>
<td>Defines the action taken when the project binds to a connected debug target. The default action is to register the application program filename, *.axf, with the connection. Then submitting a RELOAD, GO, or STEP command automatically loads the application program. Use this setting to force the application program to load as soon as the project binds to the connection.</td>
</tr>
<tr>
<td>Auto_close</td>
<td>Forces the project to close when the last connection terminates, and that project is bound to the connection.</td>
</tr>
</tbody>
</table>
When the project is version controlled using the Lock value (see Table B-1 on page B-4) use this setting to specify what action to take when the project settings are to be edited.

By default, RealView Debugger prompts to checkout the file for editing. You can change this so that the file is automatically checked out, or so that no action is taken.

When the project is version controlled using the Lock value (see Table B-1 on page B-4) use this setting to specify what action to take when the project settings are saved and the Project Properties window closes. By default, RealView Debugger prompts to check the file back into your version control software. You can change this so that the file is automatically checked in, or so that no action is taken.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vers_checkout</td>
<td>When the project is version controlled using the Lock value (see Table B-1 on page B-4) use this setting to specify what action to take when the project settings are to be edited. By default, RealView Debugger prompts to checkout the file for editing. You can change this so that the file is automatically checked out, or so that no action is taken.</td>
</tr>
<tr>
<td>Vers_checkin</td>
<td>When the project is version controlled using the Lock value (see Table B-1 on page B-4) use this setting to specify what action to take when the project settings are saved and the Project Properties window closes. By default, RealView Debugger prompts to check the file back into your version control software. You can change this so that the file is automatically checked in, or so that no action is taken.</td>
</tr>
</tbody>
</table>
B.3.2 Image_load group

The Image_load group, shown in Figure B-4, contains commands submitted to RealView Debugger when the project application is loaded to the debug target. Settings in this group override any actual values included in the image.

![Figure B-4 Image_Load group](image)

Table B-5 describes the settings available in the Image_load group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load_act</td>
<td>Controls actions on a full image download, that is, where you load the image and the symbols, and not a reload. By default, an image load disables interrupts to enable the application initialization code to establish startup conditions before accepting any interrupts. This can be disabled using this setting. You can also use this setting to reset the processor immediately following the load.</td>
</tr>
<tr>
<td>Set_pc</td>
<td>The PC is normally set on a full download, Image+Symbols, or when loading symbols only. Use this setting to override this default. You can also define a specific location for the PC on loading the image, using the Default_pc setting.</td>
</tr>
<tr>
<td>Default_pc</td>
<td>If no entry point is defined, the PC is set based on the debug target processor. This setting enables you to specify the location of the PC when no starting point is given.</td>
</tr>
</tbody>
</table>
Load_reg_set  Image loading sets certain registers by default, based on processor settings. This setting enables you to specify other registers, including memory mapping, that are set before loading the image.
Right-click and select Make New... from the context menu to enter a new register setting.
Press Enter to see the new register added at the top of the list. If multiple pathnames are given, select the Manage List... option from the context menu to control how the different paths are accessed.

Restart_act  By default, submitting the RESTART command sets the PC to the entry point of the application.
Some programs cannot be restarted in this way because they use initialized data, so use this setting to submit a reload instead of a RESTART. You can specify that it reloads Data or Code+Data.

Restart_reset  Used to reset the debug target processor prior to a restart as specified by the Restart_act setting. This is set to False by default.

Verify  By default, RealView Debugger verifies an image load by reading back the four, or eight, bytes marking the beginning and end of each section or 2k download block. Use this setting to make this verification stricter or to disable load verification completely.

Verify_warns  When a load verification fails, you can set RealView Debugger either to issue a warning, or for the load to fail.
The default is to fail the load, that is False.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load_reg_set</td>
<td>Image loading sets certain registers by default, based on processor settings. This setting enables you to specify other registers, including memory mapping, that are set before loading the image. Right-click and select Make New... from the context menu to enter a new register setting. Press Enter to see the new register added at the top of the list. If multiple pathnames are given, select the Manage List... option from the context menu to control how the different paths are accessed.</td>
</tr>
<tr>
<td>Restart_act</td>
<td>By default, submitting the RESTART command sets the PC to the entry point of the application. Some programs cannot be restarted in this way because they use initialized data, so use this setting to submit a reload instead of a RESTART. You can specify that it reloads Data or Code+Data.</td>
</tr>
<tr>
<td>Restart_reset</td>
<td>Used to reset the debug target processor prior to a restart as specified by the Restart_act setting. This is set to False by default.</td>
</tr>
<tr>
<td>Verify</td>
<td>By default, RealView Debugger verifies an image load by reading back the four, or eight, bytes marking the beginning and end of each section or 2k download block. Use this setting to make this verification stricter or to disable load verification completely.</td>
</tr>
<tr>
<td>Verify_warns</td>
<td>When a load verification fails, you can set RealView Debugger either to issue a warning, or for the load to fail. The default is to fail the load, that is False.</td>
</tr>
</tbody>
</table>
B.3.3 Auto_Set_Breaks group

The Auto_Set_Breaks group, shown in Figure B-5, contains third-level groups giving details of the breakpoints that are set automatically for the project. These breakpoints are set as soon as a symbol is matched, or on any load if there is no specified symbol. You can set these breakpoints at exit points or on special error handlers.

A Default breakpoint is provided for you to use. However, you can create new breakpoints with more meaningful names.

![Project Properties](image)

Figure B-5 Auto_Set_Breaks group

Use a third-level group for each breakpoint. You can create a new group, for example MY_BREAK_1, to store details of a personalized breakpoint. To do this, right-click on the Default group and select Make New... from the menu. You can then specify a name for the new group.

Where you want to modify an existing breakpoint, select Make Copy... from the menu. This provides starting values that can then be customized as required.

You can prompt to set selected breakpoints from the list when the project image is loaded.
Table B-6 describes the settings available in the `Auto_Set_Breaks` group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| Symbol   | Specifies the symbol name to be matched. The symbol is looked up after each load and, if matched, the breakpoint is set. This setting is optional.  
  If no symbol is specified, the breakpoint is set when the connection is established.  
  Setting the breakpoint can be optionally prompted using the `Prompt` setting. |
| Cnd      | Defines the breakpoint command to be submitted to RealView Debugger, for example `bi my_location`.  
  Setting the breakpoint can be optionally prompted. |
| Description | This setting is optional. |
| Prompt   | Indicates that RealView Debugger is to prompt for confirmation before setting the breakpoint. |
B.3.4 Named_Breaks group

The Named_Breaks group, shown in Figure B-6, contains third-level groups giving details of the named breakpoints that are set automatically for the project. Named breakpoints are breakpoints that you want to set often as part of your debugging session.

A Default breakpoint is provided for you to use. However, you can create new breakpoints with more meaningful names.

If you have specified named breakpoints for the project, these are available through the Debug → Simple Breakpoints → Named... menu for rapid deployment. Where a named breakpoint has a symbol specified, but which is not currently loaded, the named break does not show in the list.

Use a third-level group for each breakpoint. You can create a new group, for example My_NAMED_BREAK_1, to store details of a personalized breakpoint. To do this right-click on the Default group and select Make New... from the menu. You can then specify a name for the new group.

Where you want to modify an existing breakpoint, select Make Copy... from the menu. This provides starting values that can then be customized as required.
Table B-7 describes the settings available in the Named Breaks group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>Specifies the symbol name to be matched. The symbol is looked up when the breakpoints list dialog box is accessed to check the validity of the breakpoint. If no symbol is specified, the breakpoint is always enabled. This setting is optional.</td>
</tr>
<tr>
<td>Cnd</td>
<td>Defines the breakpoint command to be submitted to RealView Debugger, for example <code>bi my_location</code>.</td>
</tr>
<tr>
<td>Description</td>
<td>This setting is optional. If provided, this text is displayed in the Named Break dialog list.</td>
</tr>
</tbody>
</table>
B.3.5  Runtime_Control group

The Runtime_Control group, shown in Figure B-7, contains settings for runtime use. The group also contains a Vectors group (see Vectors group on page B-20 for details).

Table B-8 describes the settings available in the Runtime_Control group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command_line</td>
<td>Specifies an optional command line for the application when it is started.</td>
</tr>
<tr>
<td>Semihosting</td>
<td>If supported by the target processor, this setting allows the target application to access the host computer through the connection. The options available are: default: Leave the semihosting state alone. auto: Auto detect semihosting, where supported. on: Force semihosting on. off: Force semihosting off.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ARM_top_memory</td>
<td>Allows the ARM semihosting mechanism to return the top of stack and base of heap. If not defined here or in the Connection Properties, the default for each tool is used (it is different between Angel, Multi-ICE, and ARMulator). If defined here, it is set in the connection to force this address base when possible.</td>
</tr>
<tr>
<td>Vector_catch</td>
<td>Used to catch possible program errors by setting breakpoints on (or otherwise trapping) the vectors. The options available are:</td>
</tr>
<tr>
<td></td>
<td>default Catch error-type vectors but leave external and software interrupts alone.</td>
</tr>
<tr>
<td></td>
<td>auto The vectors are set only if the program does not write them in on load. This is not available on all processors.</td>
</tr>
<tr>
<td></td>
<td>on Force vector catching on.</td>
</tr>
<tr>
<td></td>
<td>off Force vector catching off.</td>
</tr>
<tr>
<td>Vector_base</td>
<td>Specifies where the vectors will be located when the processor allows them to be moved.</td>
</tr>
</tbody>
</table>
Vectors group

The Vectors group, shown in Figure B-8, contains settings to define the state of individual vectors for catching.

Table B-9 describes the settings available in the Vectors group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Set to True to catch Reset vectors. This is the default.</td>
</tr>
<tr>
<td>Undefined</td>
<td>Set to True to catch Undefined/Illegal Instructions. This is the default.</td>
</tr>
<tr>
<td>SWI</td>
<td>Set to True to catch software interrupts. The default is False.</td>
</tr>
<tr>
<td>P_Abort</td>
<td>Set to True to catch Prefetch abort (instruction fetch memory fault) exceptions. This is the default.</td>
</tr>
<tr>
<td>D_Abort</td>
<td>Set to True to catch Data abort (data access memory fault) exceptions. This is the default.</td>
</tr>
<tr>
<td>Address</td>
<td>Set to True to catch Address exceptions. This is the default.</td>
</tr>
</tbody>
</table>

*Note*  
Semihosting might set this.
Table B-9 Vectors group (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>External</td>
<td>Set to True to catch normal interrupts. The default is False.</td>
</tr>
<tr>
<td>Fast_external</td>
<td>Set to True to catch fast interrupts. The default is False.</td>
</tr>
<tr>
<td>Error</td>
<td>Set to True to catch Errors. This is the default.</td>
</tr>
</tbody>
</table>

Note
For details on how these settings map to non-ARM processors, see the documentation for your specific processor.
B.4 CONFIGURATION group

The CONFIGURATION group, shown in Figure B-9, contains settings that enable you to build your application program in different ways. This group defines the target configurations used in the build model. The most common target configurations are a Debug build, with debug information enabled and no code optimization, and a Release build, with less debug info and optimization level 2.

This group can also be used to set up different optimization levels, for example a DebugRel configuration with optimization level 1, or multiple variants of your application, for example using different device drivers. See COMPILE group on page B-24 for more details on setting different levels.

After the target configurations are defined, they are used as entries in other group in the properties settings file, for example the COMPILE=arm group.

The fields in this group are preloaded or set by default, but they can be modified as required.
Table B-10 describes the settings available in the **CONFIGURATION** group.

**Table B-10 CONFIGURATION group**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config</td>
<td>Enables you to set up the project target configurations. There must be at least one configuration. Right-click and select <strong>Make New...</strong> from the context menu to enter a new target configuration, for example <strong>Profiled</strong>. By default, a standard project sets up the <strong>Debug</strong>, <strong>DebugRel</strong>, and <strong>Release</strong> target configurations. The first entry in the list is the configuration built and loaded by default.</td>
</tr>
<tr>
<td>Active_config</td>
<td>Specifies the target configuration to build and load for the project. Right-click to see the available values as defined by the <strong>Config</strong> list.</td>
</tr>
<tr>
<td>Subdir_rule</td>
<td>By default, object files and the executable file built by the project are located in a subdirectory of the project base directory. This is given the same name as the target configuration, for example <strong>...\Debug</strong>. Use this setting to change this subdirectory rule. Right-click to see the available values. If you do not use a subdirectory, all object files for all configurations are saved in the same location and so RealView Debugger assumes that all configurations build the same objects.</td>
</tr>
</tbody>
</table>
B.5 COMPILE group

The COMPILE group contains settings that define the compilation stage of the build model for the project.

For the ARM compilers there are separate COMPILE groups to specify settings and source files:

- arm The ARM C compiler.
- arm_cpp The ARM C++ compiler.
- thumb The Thumb C compiler.

If you have other compilers, your COMPILE groups might be different.

You can set up multiple COMPILE groups to build sources that have different requirements. You can also disable some groups if required. The order in which files are processed is defined by the order of the groups, and dependencies.

In addition, you can include or exclude source files from a build by specifying values within the second-level Sources group for a chosen COMPILE group, see Table B-12 on page B-28.

By default, the COMPILE group contains the following sub-groups:

- Top-level COMPILE group settings on page B-25
- Sources group on page B-27
- Preprocessor group, ARM-specific on page B-29
- Listings group, ARM-specific on page B-30
- Messages group, ARM-specific on page B-32
- Compilation group, ARM-specific on page B-36
- Optimization group, ARM-specific on page B-42.
B.5.1 Top-level COMPILE group settings

The COMPILE group, shown in Figure B-10, contains settings that enable you to specify compiler options, source files, and other options for the C and C++ compilers you are using.

![Figure B-10 Top level COMPILE settings](image)
Table B-11 describes the settings available in the COMPILE group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| Disable   | Removes or includes a COMPILE group from a build. You can disable the group in one build configuration and enable the group in another build configuration.  
Set this to False to add the group into the build. This is the default.  
Set this to True to remove the group from the build. |
| Obj_location | Defines the location for object files relative to sources for this build.  
By default, object files are stored in the project base directory, \local\.  
However, you can use this setting to store these files in a different location.  
This is used in conjunction with the Obj_sub setting.  
Specified subdirectories are created automatically if they do not exist.  
Right-click to see the available options. |
| Obj_sub   | When the Obj_location setting has been set to sub_dir, then this setting is used to specify the subdirectory to be used.  
If the Obj_location setting has been set to sub_dir, but you do not specify the Obj_sub value, the \objects subdirectory is used by default. |
| Extra_args | Specifies the command-line arguments to the compiler that are not available through the settings interface. |
| File_args | This sets the compiler option -via to specify a file containing additional compiler arguments. For more information see the RealView Compilation Tools Compiler and Libraries Guide. |
| Tool_path | By default, the project toolchain is used to define the program used as a compiler.  
This can be overridden in the PROJECT group using the Tool_directory setting (see Table B-1 on page B-4).  
Use this setting to specify a different compiler for the project. This overrides both the project toolchain and the Tool_directory setting. |
B.5.2 Sources group

The Sources group, shown in Figure B-11, contains a list of the source files to be compiled for this COMPILE group. In this example the Sources group contains entries for two source files. To add source files to your project, do one of the following:

- right-click on the Files entry and select Edit Value, or Edit as Filename from the context menu
- right-click on the Files entry and select Manage List... from the context menu to display the Settings: List Manager dialog box
- select Add Files to Project... from the Project menu in the Code window.

![Figure B-11 Sources group](image)
Table B-12 describes the settings available in the Sources group.

**Table B-12 Sources group**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| Files | This setting specifies source files compiled for this COMPILE group. There is a separate *Files* entry for each source file in the project. This entry must only contain C or C++ source files, that is *.c or *.cpp.*   
The first entry in the list is the first file compiled.   
Right-click and select **Compile File...** from the context menu to compile the chosen file and generate a new object file, if required.   
Right-click and select **Open File in Code Window...** from the context menu to open the chosen file in the default File Editor pane ready for editing.   
Right-click and select **Exclude this file from Build** from the context menu to exclude the chosen file from this build variant. The file can be included again in the same way.   
If multiple files are given, you can select the **Manage List...** option from the context menu to reorder, add or remove entries. |
B.5.3 Preprocessor group, ARM-specific

The Preprocessor group, shown in Figure B-12, contains settings that control the operation of the compiler preprocessor.

![Figure B-12 Preprocessor group](image)

Table B-13 lists the compiler options that you can set through the Preprocessor group. For information on these settings see the *RealView Compilation Tools Compilers and Libraries Guide*.

### Table B-13 Preprocessor group

<table>
<thead>
<tr>
<th>Name</th>
<th>ARM Compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include</td>
<td>-I</td>
</tr>
<tr>
<td></td>
<td>The first entry in the list is the first location searched.</td>
</tr>
<tr>
<td>Include_rules</td>
<td>-fk</td>
</tr>
<tr>
<td>Define</td>
<td>-Dsymbol</td>
</tr>
<tr>
<td></td>
<td>and</td>
</tr>
<tr>
<td></td>
<td>-Dsymbol=value</td>
</tr>
<tr>
<td></td>
<td>The first entry in the list is the first symbol defined.</td>
</tr>
<tr>
<td>Undefine</td>
<td>-Usymbol</td>
</tr>
<tr>
<td></td>
<td>The first entry in the list is the first symbol undefined.</td>
</tr>
</tbody>
</table>
B.5.4 Listings group, ARM-specific

The Listings group, shown in Figure B-13, contains settings that control the output listings generated by the compiler for the project.

![Figure B-13 Listings group](image)

Table B-14 lists the compiler options that you can set through the Listings group. For information on these settings see the *RealView Compilation Tools Compilers and Libraries Guide*.

<table>
<thead>
<tr>
<th>Name</th>
<th>ARM Compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interlist</td>
<td>-list</td>
</tr>
</tbody>
</table>
### Table B-14 Listings group (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>ARM Compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source_expans</td>
<td><code>-list -fu</code>&lt;br&gt;Setting this to disabled is the equivalent of specifying <code>-fu</code> with <code>-list</code>.</td>
</tr>
<tr>
<td>List_includes</td>
<td>The compiler options for the available settings are:&lt;br&gt;None default&lt;br&gt;local <code>-fi</code>&lt;br&gt;global <code>-fj</code>&lt;br&gt;local and global <code>-fi -fj</code></td>
</tr>
<tr>
<td>Error_file</td>
<td><code>-errors filename</code>&lt;br&gt;The file is overwritten each time the project is recompiled.</td>
</tr>
</tbody>
</table>
B.5.5 Messages group, ARM-specific

The Messages group contains settings that control the error and warning messages generated by the compiler for the project. The default behavior is the same as the default behavior of the compiler for the variant of C or C++ specified Source Language setting of the Checking group (see Table B-18 on page B-38).

See also the Checking group on page B-38 for options that control additional diagnostic checks.

The Messages group contains:
- Warning group
- Error group on page B-35.

Warning group

The Warning group, shown in Figure B-14 on page B-33, sets the ARM compiler options that control warning messages issued during compilation of the project. All compiler warning options are available through the interface. By default, all options are set to the tool default.
Figure B-14 COMPILE Messages Warning group

Table B-15 lists the compiler warning options that you can set through the Warning group. For information on these settings see the RealView Compilation Tools Compilers and Libraries Guide.

<table>
<thead>
<tr>
<th>Name</th>
<th>ARM Compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suppress_Warnings</td>
<td>-W</td>
</tr>
<tr>
<td>Equal_sign_use</td>
<td>-Wa</td>
</tr>
<tr>
<td>Bit_field_type</td>
<td>-Wb</td>
</tr>
<tr>
<td>Deprecated_decl</td>
<td>-Wd</td>
</tr>
<tr>
<td>Deprecated_function</td>
<td>-We</td>
</tr>
<tr>
<td>Inven_ext_int</td>
<td>-Wf</td>
</tr>
</tbody>
</table>

Table B-15 Warning group
### Table B-15 Warning group (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>ARM Compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>_hdr_not_guard</td>
<td>-Wg</td>
</tr>
<tr>
<td>Implicit_constr</td>
<td>-Wl</td>
</tr>
<tr>
<td>_dbl_const_to_float</td>
<td>-Wk</td>
</tr>
<tr>
<td>Lower_prec</td>
<td>-W1</td>
</tr>
<tr>
<td>Multi_char</td>
<td>-Wm</td>
</tr>
<tr>
<td>Implicit_narrow</td>
<td>-Wn</td>
</tr>
<tr>
<td>Implicit_long_long</td>
<td>-Wo</td>
</tr>
<tr>
<td>Non_ANSI_inc</td>
<td>-Wp</td>
</tr>
<tr>
<td>Init_order</td>
<td>-Wq</td>
</tr>
<tr>
<td>Implicit_virt</td>
<td>-Wr</td>
</tr>
<tr>
<td>Struct_padding</td>
<td>-Ws</td>
</tr>
<tr>
<td>Unused_this</td>
<td>-Wt</td>
</tr>
<tr>
<td>Future_compat</td>
<td>-Wu</td>
</tr>
<tr>
<td>Implicit_return</td>
<td>-Wv</td>
</tr>
<tr>
<td>Fn_not_used</td>
<td>-WX</td>
</tr>
<tr>
<td>Deprecated_features</td>
<td>-Wy</td>
</tr>
</tbody>
</table>
Error group

The Error group, shown in Figure B-15, sets the ARM compiler options that control error messages issued during compilation of the project. All compiler error options are available through the interface. By default, all options are set to the tool default.

Table B-16 lists the compiler error options that you can set through the Error group. For information on these settings see the RealView Compilation Tools Compilers and Libraries Guide.

<table>
<thead>
<tr>
<th>Name</th>
<th>ARM Compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downgr_ac_errs</td>
<td>-Ea</td>
</tr>
<tr>
<td>Implicit_casts</td>
<td>-Ec</td>
</tr>
<tr>
<td>Unclean_casts</td>
<td>-Ef</td>
</tr>
<tr>
<td>Downgr_cn_errs</td>
<td>-Ei</td>
</tr>
<tr>
<td>Linkage</td>
<td>-Ei</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>-Ep</td>
</tr>
<tr>
<td>Zero_len_array</td>
<td>-Ez</td>
</tr>
</tbody>
</table>

Figure B-15 COMPILE Messages Error group
B.5.6 Compilation group, ARM-specific

The Compilation group contains settings that control the C and C++ compilers.

The Compilation group contains:
- Compilation group settings
- Checking group on page B-38
- APCS group on page B-40
- Alignment group on page B-41.

Compilation group settings

The Compilation group, shown in Figure B-16, contains a series of settings that describe project compilation options.

![Figure B-16 COMPILE Compilation group](image)
Table B-17 lists the compiler options that you can set through the Compilation group. For information on these settings see the *RealView Compilation Tools Compilers and Libraries Guide*.

<table>
<thead>
<tr>
<th>Name</th>
<th>ARM Compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>The options for the available settings are:</td>
</tr>
<tr>
<td>ARM_C</td>
<td>armcc</td>
</tr>
<tr>
<td>Thumb_C</td>
<td>tcc</td>
</tr>
<tr>
<td>ARM_Cpp</td>
<td>armcpp</td>
</tr>
<tr>
<td>Thumb_Cpp</td>
<td>tcpp</td>
</tr>
<tr>
<td>Endianness</td>
<td>-littleend</td>
</tr>
<tr>
<td></td>
<td>and</td>
</tr>
<tr>
<td></td>
<td>-bigend</td>
</tr>
<tr>
<td>Generate_debug</td>
<td>-g</td>
</tr>
<tr>
<td>Debug_format</td>
<td>-dwarf2</td>
</tr>
<tr>
<td>ELF_section_per_fn</td>
<td>-zo</td>
</tr>
<tr>
<td>Char</td>
<td>-zc</td>
</tr>
<tr>
<td>Enums_as_ints</td>
<td>-fy</td>
</tr>
<tr>
<td>Fp_processing</td>
<td>-fpu</td>
</tr>
</tbody>
</table>

--- **Note** ---

Some options are not valid for the Thumb compiler. Make sure you choose valid options, otherwise, the files might not compile. See the documentation for the compiler you are using.
Checking group

The Checking group, shown in Figure B-17, controls general compiler actions, including making certain checking operations stricter as an aid to portability and good coding practise.

By default, all settings are set to the default for the variant of C used to construct the source files for the project. This is defined by setting the Source language setting.

![Figure B-17 COMPILE Compilation Checking group](image)

Table B-18 lists the compiler options that you can set through the Checking group. For information on these settings see the RealView Compilation Tools Compilers and Libraries Guide.

<table>
<thead>
<tr>
<th>Name</th>
<th>ARM Compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source_language</td>
<td>The compiler options for the available settings are:</td>
</tr>
<tr>
<td></td>
<td>default default</td>
</tr>
<tr>
<td></td>
<td>ANSI -ansi</td>
</tr>
<tr>
<td></td>
<td>Strict -strict</td>
</tr>
<tr>
<td></td>
<td>EmbeddedCPlusPlus -embeddedcplusplus</td>
</tr>
<tr>
<td>Data_flow</td>
<td>-fa</td>
</tr>
<tr>
<td>Obj_declar</td>
<td>-fh</td>
</tr>
</tbody>
</table>
Table B-18 Checking group (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>ARM Compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit_casts</td>
<td>-fp</td>
</tr>
<tr>
<td>Unused_declar</td>
<td>-fv</td>
</tr>
<tr>
<td>Enable_suppressed</td>
<td>-fx</td>
</tr>
</tbody>
</table>
APCS group

The APCS group, shown in Figure B-18, controls ARM Procedure Call Standard (APCS) qualifiers. For full details see ARM/Thumb® Procedure Call Standard (ATPCS) Specification.

Table B-19 lists the compiler options that you can set through the APCS group. For information on these settings see the RealView Compilation Tools Compilers and Libraries Guide.

<table>
<thead>
<tr>
<th>Name</th>
<th>ARM Compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack_checking</td>
<td>-[no]swstackcheck</td>
</tr>
<tr>
<td>Interworking</td>
<td>-[no]interwork</td>
</tr>
<tr>
<td>Ropi</td>
<td>-[no]ropi</td>
</tr>
<tr>
<td>Rwpi</td>
<td>-[no]rwpi</td>
</tr>
</tbody>
</table>
Alignment group

The Alignment group, shown in Figure B-19, controls alignment options in the compiled code.

Table B-20 lists the compiler options that you can set through the Alignment group. For information on these settings see the RealView Compilation Tools Compilers and Libraries Guide.

<table>
<thead>
<tr>
<th>Name</th>
<th>ARM Compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Struct_align</td>
<td>-zasNumber</td>
</tr>
</tbody>
</table>

Right-click to see the available options.
B.5.7 Optimization group, ARM-specific

The Optimization group, shown in Figure B-20, controls how aggressively the compiler tries to improve the machine code it generates and enables you to disable some optimizations.

![Figure B-20 COMPILE Optimization group](image)
Table B-21 lists the compiler options that you can set through the Optimization group. For information on these settings see the RealView Compilation Tools Compilers and Libraries Guide.

Table B-21 Optimization group

<table>
<thead>
<tr>
<th>Name</th>
<th>ARM Compiler option</th>
</tr>
</thead>
</table>
| Speed_vs_space| The options for the available settings are:  
|               | default -Ospace     |
|               | speed -Otime        |
|               | space -Ospace       |
| Inline        | The options for the available settings are:  
|               | disabled -Ono_inline|
|               | enabled -Oinline    |
| Auto_inline   | The options for the available settings are:  
|               | default -Oautoinline|
|               | enabled -Oautoinline|
|               | disabled -Ono_autoinline|
| Ldrd          | The options for the available settings are:  
|               | disabled -Ono_lrd   |
|               | enabled -Olrd       |
| No_data_reorder| The options for the available settings are:  
|                | disabled -Ono_data_reorder|
|                | enabled -Odata_reorder|
| Cpu           | -cpu                |
| Debug_optimize| The options for the available settings are:  
|               | none -O0            |
|               | partial -O1         |
|               | full -O2            |
B.6 ASSEMBLE group

The ASSEMBLE group contains settings that define the assembly stage of the build model for the project.

You can set up multiple ASSEMBLE groups to build sources that have different requirements. You can also disable some groups if required. The order in which files are processed is defined by the order of the groups, and dependencies.

In addition, files can be included or excluded from a build by settings in the second-level Sources group for a chosen ASSEMBLE group (see Table B-23 on page B-48).

By default, the ASSEMBLE group contains:

- *Top-level ASSEMBLE group* on page B-45
- *Sources group* on page B-47
- *Preprocessor group, ARM-specific* on page B-49
- *Listings group, ARM-specific* on page B-50
- *Messages group, ARM-specific* on page B-52
- *Assembly group, ARM-specific* on page B-53.
B.6.1 Top-level ASSEMBLE group

The ASSEMBLE group, shown in Figure B-21, contains settings that control assembler tool invocation and object location.

![Project Properties](image)

Figure B-21 Top level ASSEMBLE group

Table B-22 describes the settings available in the ASSEMBLE group. For information on these settings see the RealView Compilation Tools Compilers and Libraries Guide,

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable</td>
<td>Removes or includes an ASSEMBLE group from a build.</td>
</tr>
<tr>
<td></td>
<td>Set this to False to add the group into the build. This is the default.</td>
</tr>
<tr>
<td></td>
<td>Setting this to True does not delete the group but it is invisible to the build.</td>
</tr>
<tr>
<td>Obj_location</td>
<td>Defines the location for object files relative to sources for this build.</td>
</tr>
<tr>
<td></td>
<td>By default, object files are stored in the project base directory, local. But you can use this setting to store these files in a different location.</td>
</tr>
<tr>
<td></td>
<td>This is used in conjunction with the Obj_sub setting.</td>
</tr>
<tr>
<td></td>
<td>Specified subdirectories are created automatically if they do not exist.</td>
</tr>
<tr>
<td>Obj_sub</td>
<td>When the Obj_location setting has been set to sub_dir, this setting is used to specify the subdirectory to be used.</td>
</tr>
<tr>
<td></td>
<td>If the Obj_location setting has been set to sub_dir, but you do not specify the Obj_sub value, the \objects subdirectory is used by default.</td>
</tr>
</tbody>
</table>
### Table B-22 ASSEMBLE group (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra_args</td>
<td>Use this field to specify command-line arguments to the assembler that are not available through the settings interface.</td>
</tr>
<tr>
<td>File_args</td>
<td>This sets the assembler option <code>-via</code> to specify a file containing additional assembler arguments.</td>
</tr>
</tbody>
</table>
| Tool_path  | By default, the project toolchain is used to define the program used as an assembler.  
This can be overridden in the PROJECT group using the Tool_directory setting (see Table B-1 on page B-4).  
Use this setting to specify a different assembler for the project. This overrides both the project toolchain and the Tool_directory setting. |
B.6.2 Sources group

The Sources group, shown in Figure B-22, contains a list of the files to be assembled for this group.

![Figure B-22 ASSEMBLE Sources group](image)

---

**Figure B-22 ASSEMBLE Sources group**
Table B-23 describes the settings available in the Sources group. For information on these settings see the *RealView Compilation Tools Assembler Guide*.

**Table B-23 Sources group**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files</td>
<td>This setting enables you to specify files assembled for this ASSEMBLE group. There is a separate *Files entry for each source file in the project. This entry must contain only assembler source files, that is, *.asm, *.s, or *.src, or others specified by your assembler. The first entry in the list is the first file assembled. Right-click and select <strong>Compile File</strong>... from the context menu to assemble the chosen file and generate a new object file, if required. Right-click and select <strong>Open File in Code Window</strong>... from the context menu to open the chosen file in the default File Editor pane ready for editing. Right-click and select <strong>Exclude this file from Build</strong> from the context menu to exclude the chosen file from this build variant. The file can be included again in the same way. If multiple files are given, you can select the <strong>Manage List</strong>... option from the context menu to reorder, add or remove entries.</td>
</tr>
</tbody>
</table>
B.6.3 Preprocessor group, ARM-specific

The Preprocessor group, shown in Figure B-23, contains settings that control the operation of the assembler when processing directives. Use these settings to control how your source files are assembled.

![Preprocessor group settings](image)

**Figure B-23 ASSEMBLE Preprocessor group**

Table B-24 describes the settings available in the Preprocessor group. For information on these settings see the *RealView Compilation Tools Assembler Guide*.

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include</td>
<td>-i</td>
</tr>
<tr>
<td>Directives</td>
<td>-pd</td>
</tr>
</tbody>
</table>

Table B-24 Preprocessor group
B.6.4 Listings group, ARM-specific

The Listings group, shown in Figure B-24, contains settings that control the output listings generated by the assembler for the project.

![Figure B-24 ASSEMBLE Listings group](image)

Table B-25 describes the settings available in the Listings group. For information on these settings see the *RealView Compilation Tools Assembler Guide*.

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing</td>
<td>-list</td>
</tr>
<tr>
<td>Listing_file</td>
<td>The filename argument to -list.</td>
</tr>
<tr>
<td>Listing_opts</td>
<td>The assembler options for the available settings: are: none</td>
</tr>
<tr>
<td>Listing_width</td>
<td>-width</td>
</tr>
</tbody>
</table>
Table B-25 Listings group (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing_length</td>
<td>-length</td>
</tr>
<tr>
<td>Dependency_list</td>
<td>-depend</td>
</tr>
<tr>
<td>Error_file</td>
<td>-errors</td>
</tr>
</tbody>
</table>
B.6.5 Messages group, ARM-specific

The Messages group, shown in Figure B-25, contains settings that control messages generated by the assembler.

![Figure B-25 ASSEMBLE Messages group](image)

Table B-26 describes the settings available in the Messages group. For information on these settings see the RealView Compilation Tools Assembler Guide.

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warnings</td>
<td>-warn</td>
</tr>
<tr>
<td>Downgr_cpu_errs</td>
<td>-unsafe</td>
</tr>
<tr>
<td>Check_reg_lists</td>
<td>-cr</td>
</tr>
<tr>
<td>Split_ldm</td>
<td>-split_ldm</td>
</tr>
</tbody>
</table>
B.6.6 Assembly group, ARM-specific

The Assembly group contains settings that control the format of the executable file built by the project.

Top-level Assembly group settings

The Assembly group, shown in Figure B-26, contains a series of settings that describe project assembler options.

Figure B-26 Assembly group

Table B-27 describes the settings available in the Assembly group. For information on these settings see the RealView Compilation Tools Assembler Guide.

Table B-27 Assembly group

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assemble_mode</td>
<td>-16, -32</td>
</tr>
<tr>
<td>Endianness</td>
<td>-bi</td>
</tr>
<tr>
<td>Debug_format</td>
<td>-dwarf2</td>
</tr>
<tr>
<td>Keep</td>
<td>-keep</td>
</tr>
<tr>
<td>Cpu</td>
<td>-cpu</td>
</tr>
</tbody>
</table>
### APCS group

The APCS group, shown in Figure B-27 on page B-54, controls ARM Procedure Call Standard (APCS) qualifiers.

---

**Table B-27 Assembly group (continued)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fp_processing</td>
<td>-fpu</td>
</tr>
</tbody>
</table>

---

**Note**

Some options are not valid for Thumb assembly code. Make sure you choose valid options, otherwise the files might not assemble. See the documentation for the assembler you are using.

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source_cache</td>
<td>-nocache</td>
</tr>
<tr>
<td>Max_cache</td>
<td>-maxcache</td>
</tr>
<tr>
<td>Ignore_special</td>
<td>-noesc</td>
</tr>
<tr>
<td>Predef_reg</td>
<td>-noregs</td>
</tr>
<tr>
<td>Generate_debug</td>
<td>-g</td>
</tr>
</tbody>
</table>

---

**Figure B-27 ASSEMBLE Assembly APCS group**
Table B-28 describes the settings available in the APCS group. For information on these settings see the *RealView Compilation Tools Assembler Guide*.

Table B-28 APCS group

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apcs</td>
<td>The assembler options for the available settings are:</td>
</tr>
<tr>
<td></td>
<td>disabled -apcs /none</td>
</tr>
<tr>
<td></td>
<td>enabled -apcs /default</td>
</tr>
<tr>
<td>Stack_checking</td>
<td>-apcs /[no]swstackcheck</td>
</tr>
<tr>
<td>Interworking</td>
<td>-apcs /[no]intwerwork</td>
</tr>
<tr>
<td>Ropi</td>
<td>-apcs /[no]ropi</td>
</tr>
<tr>
<td>Rwpi</td>
<td>-apcs /[no]rwpi</td>
</tr>
<tr>
<td>Swstna</td>
<td>-apcs /swstna</td>
</tr>
</tbody>
</table>
B.7 CUSTOM group

The CUSTOM group, shown in Figure B-28, contains a series of settings that describe custom project options. These settings define how to build:

- headers, for example .h files
- sources, for example .c, .asm, or .cpp files
- object files, that is .o or .obj files
- libraries.

You can also use a CUSTOM group to specify tools that are run by other CUSTOM groups, or for use in prelinker and postlinker operations.

Each CUSTOM group you create can build one or more files. The files listed under the Files value are the explicit output. So if a source and header are produced, only one is listed, and it must be the one that something else is dependent on.

If multiple files are listed using the Files value, the custom command specified is run for each file listed.

Each time the project is built, the date and time of the data file are compared against the output file and if the data is newer, the output file is rebuilt.

The default group CUSTOM=default contains an empty Custom group. If you want to build your own files, create a new group by right-clicking and selecting Make New... from the context menu. You can also select Make Copy... and then modify existing settings from the default group.
You can set up several CUSTOM groups to build different types of files. The order in which files are processed is defined by the order of the groups and dependencies.

Table B-29 describes the settings available in the CUSTOM group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable</td>
<td>Removes or includes a CUSTOM group from a build. Set this to False to add the group into the build. This is the default. Setting this to True does not delete the group, but it is invisible to the build.</td>
</tr>
<tr>
<td>Description</td>
<td>Describes the tool to use in the custom build.</td>
</tr>
<tr>
<td>Message</td>
<td>Contains the message that is displayed when the custom command is run.</td>
</tr>
<tr>
<td>Files</td>
<td>Specifies the main output file from the custom command. If the command produces more than one file then this entry must list only one of them. Use multiple Files entries only if you want the command run once for each of the files. Use a fake filename, that is one that is not produced, if you want the command to run every time.</td>
</tr>
<tr>
<td>Depends_on</td>
<td>Specifies the files used as input data to the build process for the given custom command. When the request to rebuild is made, the data and time of this file is checked to see if it requires rebuilding. If any other dependent file in the list is newer, it is also rebuilt.</td>
</tr>
<tr>
<td>Command</td>
<td>Specifies the host command to run to produce the required output file or to perform the required operation. The command can include macros, for example $@ for the output file or $? to list dependent files that are newer. You can use any valid commands, depending on your makefile. Commands that use shell commands, for example echo and for, must be preceded by a plus sign. Using an at sign before the command prevents echoing the command before it is run.</td>
</tr>
<tr>
<td>Use_as</td>
<td>Specifies when and how the output files must be rebuilt.</td>
</tr>
</tbody>
</table>
B.8 BUILD group

The BUILD group contains settings that link the object files created by the COMPILE, ASSEMBLE, and CUSTOM groups into an application program. This group invokes the linker after all other files have been checked for rebuild, so you can only have one BUILD group for a project.

The BUILD group also enables you to use prelink and postlink commands. By default, the BUILD group contains:

- Top-level BUILD group settings on page B-59
- Listings group, ARM-specific on page B-61
- Messages group, ARM-specific on page B-62
- Link_Advanced group, ARM-specific on page B-63
- Symbol_Control group, ARM-specific on page B-65
- Pre_Post_Link group on page B-67
- RVDEBUG_Commands group on page B-68.
B.8.1 Top-level BUILD group settings

The BUILD group, shown in Figure B-29, contains a series of settings that describe project settings.

![Figure B-29 BUILD group](image-url)
Table B-30 describes the settings available in the BUILD group. For information on these settings see the RealView Compilation Tools Linker and Utilities Guide.

<table>
<thead>
<tr>
<th>Name</th>
<th>Linker option or description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>This setting names the output from the linker. If no path is given, the named file is saved in the project base directory as defined by the default target configuration.</td>
</tr>
<tr>
<td>Objects</td>
<td>This is equivalent to listing a specific object file on the command line that invokes the linker. Use this setting to define object files that you want to link in to your project but which are:</td>
</tr>
<tr>
<td></td>
<td>• not built by this project</td>
</tr>
<tr>
<td></td>
<td>• not part of a library referenced from this project. The interface displays an <code>**Objects**</code> instance for each object you reference.</td>
</tr>
<tr>
<td>Lib_paths</td>
<td><code>-libpath</code></td>
</tr>
<tr>
<td>Libraries</td>
<td>This setting defines the location of a library that is not in the normal library search path defined by the <code>Lib_paths</code> setting and the <code>ARMLIB</code> environment variable.</td>
</tr>
<tr>
<td>Extra_args</td>
<td>Specifies the command-line arguments to the linker that are not available through the settings interface.</td>
</tr>
<tr>
<td>File_args</td>
<td>This sets the linker option <code>-via</code> to specify a file containing additional linker arguments.</td>
</tr>
<tr>
<td>Tool_path</td>
<td>By default, the project toolchain is used to define the program used as a linker. This can be overridden in the PROJECT group using the <code>Tool_directory</code> setting (see Table B-1 on page B-4). Use this setting to specify a different linker for the project. This overrides both the project toolchain and the <code>Tool_directory</code> setting.</td>
</tr>
<tr>
<td>Makefile</td>
<td>Specifies the makefile built for the project.</td>
</tr>
<tr>
<td>Make_template</td>
<td>Specifies the filename of a makefile template to use when creating the project makefiles. You can use this to customize the heading comments and the variables available to commands in the makefile.</td>
</tr>
<tr>
<td>Debug_info</td>
<td><code>-nodebug</code></td>
</tr>
</tbody>
</table>
B.8.2 Listings group, ARM-specific

The Listings group, shown in Figure B-30, contains settings that control the output listings generated by the linker for the project.

![Figure B-30 BUILD Listings group](image)

Table B-31 describes the settings available in the Listings group. For information on these settings see the *RealView Compilation Tools Linker and Utilities Guide*.

<table>
<thead>
<tr>
<th>Name</th>
<th>Linker option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol_file</td>
<td>-symdefs</td>
</tr>
<tr>
<td>Listing_file</td>
<td>-list</td>
</tr>
<tr>
<td>Error_file</td>
<td>-errors</td>
</tr>
</tbody>
</table>
B.8.3 Messages group, ARM-specific

The Messages group, shown in Figure B-31, contains settings that control messages displayed during the linking stage.

![Figure B-31 BUILD Messages group](image)

Table B-32 describes the settings available in the Messages group. For information on these settings see the *RealView Compilation Tools Linker and Utilities Guide*.

<table>
<thead>
<tr>
<th>Name</th>
<th>Linker option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sizes</td>
<td>The linker options for the available settings are:</td>
</tr>
<tr>
<td></td>
<td>None default</td>
</tr>
<tr>
<td></td>
<td>details -info sizes</td>
</tr>
<tr>
<td></td>
<td>totals -info totals</td>
</tr>
<tr>
<td></td>
<td>both -info sizes -info totals</td>
</tr>
<tr>
<td>Veneers</td>
<td>-info veneers</td>
</tr>
<tr>
<td>Unused_sections</td>
<td>-info unused</td>
</tr>
<tr>
<td>Progress</td>
<td>-v</td>
</tr>
<tr>
<td>Callgraph</td>
<td>-callgraph</td>
</tr>
</tbody>
</table>
B.8.4 Link_Advanced group, ARM-specific

The Link_Advanced group, shown in Figure B-32, contains settings that control advanced linker commands.

![Figure B-32 BUILD Link_Advanced group](image)

Table B-33 describes the settings available in the Link_Advanced group. For information on these settings see the RealView Compilation Tools Linker and Utilities Guide.

<table>
<thead>
<tr>
<th>Name</th>
<th>Linker option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td>-entry</td>
</tr>
<tr>
<td>Scatter_file</td>
<td>-scatter</td>
</tr>
<tr>
<td>Relocatable</td>
<td>-reloc</td>
</tr>
<tr>
<td>Split</td>
<td>-split</td>
</tr>
<tr>
<td>Ro_base</td>
<td>-ro-base</td>
</tr>
<tr>
<td>Ropi</td>
<td>-ropi</td>
</tr>
<tr>
<td>Rw_base</td>
<td>-rw-base</td>
</tr>
</tbody>
</table>
The linker options for the available settings are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Linker option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rwpi</td>
<td>-rwpi</td>
</tr>
<tr>
<td>Keep</td>
<td>-keep</td>
</tr>
<tr>
<td>First</td>
<td>-first</td>
</tr>
<tr>
<td>Last</td>
<td>-last</td>
</tr>
<tr>
<td>Remove_unused</td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>linker default</td>
</tr>
<tr>
<td>disabled</td>
<td>-noremove</td>
</tr>
<tr>
<td>enabled</td>
<td>-remove</td>
</tr>
<tr>
<td>Partial</td>
<td>-partial</td>
</tr>
<tr>
<td>Strict</td>
<td>-strict</td>
</tr>
<tr>
<td>Entry_point</td>
<td>-entry</td>
</tr>
</tbody>
</table>
B.8.5 Symbol_Control group, ARM-specific

The Symbol_Control group, shown in Figure B-33, controls how symbols and space are handled during the build stage.

![Figure B-33 BUILD Symbol_Control group](image-url)
Table B-34 describes the settings available in the Symbol_Control group. For information on these settings see the *RealView Compilation Tools Linker and Utilities Guide*.

<table>
<thead>
<tr>
<th>Name</th>
<th>Linker option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locals</td>
<td>The linker options for the available settings are:</td>
</tr>
<tr>
<td></td>
<td>add -locals</td>
</tr>
<tr>
<td></td>
<td>remove -nolocals</td>
</tr>
<tr>
<td>Sect_xref</td>
<td>The linker options for the available settings are:</td>
</tr>
<tr>
<td></td>
<td>none linker default</td>
</tr>
<tr>
<td></td>
<td>from -xreffrom</td>
</tr>
<tr>
<td></td>
<td>to -xrefto</td>
</tr>
<tr>
<td></td>
<td>both -xreffrom -xrefto</td>
</tr>
<tr>
<td>Symbols</td>
<td>The linker options for the available settings are:</td>
</tr>
<tr>
<td></td>
<td>disabled default</td>
</tr>
<tr>
<td></td>
<td>enabled -symbols</td>
</tr>
<tr>
<td>Mangled</td>
<td>The linker options for the available settings are:</td>
</tr>
<tr>
<td></td>
<td>Default linker default</td>
</tr>
<tr>
<td></td>
<td>unmangled -umangled</td>
</tr>
<tr>
<td></td>
<td>mangled -mangled</td>
</tr>
<tr>
<td>Memory_map</td>
<td>Generates a memory map listing when enabled. The default is disabled.</td>
</tr>
<tr>
<td>Xref</td>
<td>Generates a listing of references between input areas when enabled. The default is disabled.</td>
</tr>
</tbody>
</table>
B.8.6 Pre_Post_Link group

The Pre_Post_Link group, shown in Figure B-34, enables you to run specific commands just before linking, or immediately after. You can insert OS commands, for example DOS/Windows, or UNIX shell commands, and any program on the local workstation. You can also use this group to insert $macro_name make commands.

If you are working in Windows, you must precede a shell command with a plus sign, for example +echo build complete.

Examine the makefile to see what macros are available for these commands.

![Figure B-34 BUILD Pre_Post_Link group](image)

Table B-35 describes the settings available in the Pre_Post_Link group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre_link</td>
<td>Enables you to specify commands to be run before linking.</td>
</tr>
<tr>
<td>Post_link</td>
<td>Enables you to specify commands to be run immediately after linking.</td>
</tr>
</tbody>
</table>
B.8.7 RVDEBUG_Commands group

The RVDEBUG_Commands group, shown in Figure B-35, contains commands that are run as part of the build.

This group contains a single setting that enables you to create a list of postbuild commands that run as soon as the build completes. They are only run if the build succeeds.

Table B-36 RVDEBUG_Commands group

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post_build</td>
<td>Enables you to specify commands to be run immediately after the build. Use %s in the command to expand the target filename.</td>
</tr>
</tbody>
</table>
B.9 BUILD_LIB group

The BUILD_LIB group is displayed when you create a Library project. It contains settings that combine the object files created by the COMPILE, ASSEMBLE, and CUSTOM groups into a library. This group invokes the librarian after all other files have been checked for rebuild, so you can only have one BUILD_LIB group for a project.

By default, the BUILD_LIB group contains:

- Top-level BUILD_LIB group on page B-70
- Listings group, ARM-specific on page B-72
- Messages group, ARM-specific on page B-73
- Pre_Post_Lib group on page B-74
- RVDEBUG_Commands group on page B-76.
B.9.1 Top-level BUILD_LIB group

The BUILD_LIB group, shown in Figure B-36, contains a series of settings that describe project settings for the librarian or archiver. For ARM tools, this is normally the program armar.

![Figure B-36 BUILD_LIB group](image)

Table B-37 describes the settings available in the BUILD_LIB group. For information on these settings see the RealView Compilation Tools Linker and Utilities Guide.

<table>
<thead>
<tr>
<th>Name</th>
<th>armar option or description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library</td>
<td>Defines the name of the target library built for the project.</td>
</tr>
<tr>
<td></td>
<td>The specified .a file is located in the project base directory as defined by the</td>
</tr>
<tr>
<td></td>
<td>default target configuration, unless a path is given.</td>
</tr>
<tr>
<td>Objects</td>
<td>Defines an object file or files to include in the library that are not included</td>
</tr>
<tr>
<td></td>
<td>as sources, for example third-party interface modules.</td>
</tr>
<tr>
<td>Extra_args</td>
<td>Specifies the command-line arguments to the librarian that are not</td>
</tr>
<tr>
<td></td>
<td>available through the settings interface.</td>
</tr>
<tr>
<td>File_args</td>
<td>This sets the option -v to specify a file containing additional arguments.</td>
</tr>
</tbody>
</table>
By default, the project toolchain is used to define the program used as a librarian.

This can be overridden in the `PROJECT` group using the `Tool_directory` setting (see Table B-1 on page B-4).

Use this setting to override these other settings for this particular library.

<table>
<thead>
<tr>
<th>Name</th>
<th>armar option or description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makefile</td>
<td>Specifies the makefile built for the project.</td>
</tr>
<tr>
<td>Make_template</td>
<td>Specifies the filename of a makefile template to use when creating the project makefiles. You can use this to customize the heading comments and the variables available to commands in the makefile.</td>
</tr>
<tr>
<td>Create_new</td>
<td>--create</td>
</tr>
</tbody>
</table>
B.9.2 Listings group, ARM-specific

The Listings group, shown in Figure B-37, contains settings that control the output listings generated by the librarian for the project.

Table B-38 describes the settings available in the Listings group. For information on these settings see the RealView Compilation Tools Linker and Utilities Guide.

<table>
<thead>
<tr>
<th>Name</th>
<th>armar option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entries</td>
<td>-entries</td>
</tr>
<tr>
<td>Contents</td>
<td>-p</td>
</tr>
<tr>
<td>Table_of_contents</td>
<td>-t</td>
</tr>
<tr>
<td>Sizes</td>
<td>-sizes</td>
</tr>
<tr>
<td>Symbol_table</td>
<td>-zs</td>
</tr>
</tbody>
</table>
B.9.3 Messages group, ARM-specific

The Messages group, shown in Figure B-38, contains settings that control messages displayed during the library creation.

![Figure B-38 BUILD_LIB Messages group](image)

Table B-39 describes the settings available in the Messages group. For information on these settings see the *RealView Compilation Tools Linker and Utilities Guide*.

<table>
<thead>
<tr>
<th>Name</th>
<th>armlib option</th>
</tr>
</thead>
<tbody>
<tr>
<td>No_creation</td>
<td>-c</td>
</tr>
<tr>
<td>Verbose</td>
<td>-v</td>
</tr>
<tr>
<td>Version</td>
<td>-vsn</td>
</tr>
</tbody>
</table>
B.9.4 Pre_Post_Lib group

The Pre_Post_Lib group, shown in Figure B-39, enables you to run specific commands just before the librarian, or immediately after. You can insert OS commands, for example DOS/Windows, or UNIX shell commands, and any program on the local workstation. You can also use this group to insert $(macro_name) make commands.

If you are working in Windows, you must precede a shell command with a plus sign, for example +echo this is a message. Examine the makefile to see what macros are available for these commands.

Figure B-39 BUILD_LIB Pre_Post_Lib group
Table B-40 describes the settings available in the Pre_Post_Lib group.

Table B-40 Pre_Post_Lib group

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre_lib</td>
<td>Enables you to specify commands to be run before the librarian. The makefile defines the following variables that can be used in this command:</td>
</tr>
<tr>
<td></td>
<td>OBJS: Space separated list of object files to insert in the library.</td>
</tr>
<tr>
<td></td>
<td>PROGRAM: The filename of the library to create or update.</td>
</tr>
<tr>
<td>Post_lib</td>
<td>Enables you to specify commands to be run immediately after the librarian. The makefile defines the following variables that can be used in this command:</td>
</tr>
<tr>
<td></td>
<td>OBJS: Space separated list of object files to insert in the library.</td>
</tr>
<tr>
<td></td>
<td>PROGRAM: The filename of the library to create or update.</td>
</tr>
</tbody>
</table>
B.9.5 RVDEBUG_Commands group

The RVDEBUG_Commands group, shown in Figure B-40, contains commands that are run as part of the build.

This group contains a single setting that enables you to create a list of post-build commands that run as soon as the build completes. They are only run if the build succeeds.

Table B-41 describes the settings available in the RVDEBUG_Commands group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post_build</td>
<td>Specifies the commands to be run immediately after the build. Use %s in the command to expand the target filename.</td>
</tr>
</tbody>
</table>
B.10 MAKEFILE group

The MAKEFILE group, shown in Figure B-41, is preloaded when you create a Custom project.

--- Note ---

For an auto-project, these settings are set up for a no-build project. Do not change the settings in this group for an auto-project.

You can create a user-defined project using the image for an auto-project, and choose to merge the auto-project settings into the user-defined project settings. The MAKEFILE group is not merged. For more details, see Merging auto-project settings into a project on page 11-36.

![Figure B-41 MAKEFILE group](Image)

The MAKEFILE group contains a series of settings that override the default makefile and enable you to use your own.

Table B-42 describes the settings available in the MAKEFILE group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makefile</td>
<td>Specifies the makefile built for the project, if this project contains a makefile.</td>
</tr>
<tr>
<td>Application</td>
<td>Defines the target application program built for the project.</td>
</tr>
</tbody>
</table>
### Table B-42 MAKEFILE group (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>Specifies the arguments to pass to the make or other build command.</td>
</tr>
<tr>
<td>Cwd</td>
<td>Specifies the working directory when running your build.</td>
</tr>
<tr>
<td>Command</td>
<td>Specifies your own make command that can be make, another build tool, or none if this is a no-build project.</td>
</tr>
<tr>
<td></td>
<td>See <em>Using your own make command</em> on page B-79 for more details.</td>
</tr>
</tbody>
</table>
B.10.1 Using your own make command

In a simple project, the default build rule is to run `make`. For a Custom project, you can run `make` or another build tool.

By default, the project toolchain defines the location of the `make` command if you do not specify your own command.

Your `make` command can include controls that are expanded by RealView Debugger. You can use this setting to create any command line you require.

Specify each control character as `$` followed by a letter as in:

```
make -f $f $a $e $t $p
```

where:

- `$f` is the name of the makefile from the Makefile setting.
- `$a` is the arguments list from the Arguments setting.
- `$e` depends on the action you select on the Tools menu:
  - `all` if you select Tools → Build...
  - `rebuild` if you select Tools → Rebuild All (Clean+Build)
  - `clean` if you select Tools → Clean (remove objects)
  - `object_file` if you select Tools → Build This File. This might include a path name that is relative to the project directory.
- `$t` is the target filename that is built, or cleaned and built from the Application setting.
- `$p` is the project directory. This is the directory you specified when you created the project.

If the Command setting is empty then the command defaults to `make -f $f $a $e`.

——— Note ————

To successfully build your Custom project, remove the `$e` control character, and use your own arguments as required.
Appendix C
RealView Debugger on Solaris and Linux

This appendix provides supplementary information for developers using RealView Debugger on Solaris and Linux. It contains the following sections:

- About this Appendix on page C-2
- Installing RealView Debugger on page C-3
- Getting more information on page C-9
- Changes to target configuration details on page C-10
- Changes to GUI and general user information on page C-16
- Connecting to remote hosts on page C-24.
C.1 About this Appendix

The RealView Debugger v1.6 documentation for Windows has been updated to include new features and enhancements found in v1.6.1. To aid readability, this release is referred to as RealView Debugger v1.6 throughout the documentation suite.

This Appendix describes features that are specific to using RealView Debugger v1.6.1 on Solaris and Linux, and contains corrections and additions to the documentation suite:

- RealView Debugger v1.6 Essentials Guide
- RealView Debugger v1.6 User Guide
- RealView Debugger v1.6 Target Configuration Guide
- RealView Debugger v1.6 Command Line Reference Guide
- RealView Debugger v1.6 Extensions User Guide.

C.1.1 Examples

The examples given in the RealView Debugger v1.6 documentation for Windows have all been tested and shown to work as described. Developers using Solaris or Linux must amend the instructions given in examples. For example, if you want to access the dhrystone example project you must look in install_directory/RVD/Examples/1.6.1/release/unix/dhrystone.

In general, examples use the ARMulator software simulator to simulate an ARM-based debug target. In some cases, examples are given for other debug target systems. Developers using Solaris or Linux must amend the instructions to use RealView ARMulator ISS.
C.2 Installing RealView Debugger

This section describes:
- Types of installation
- Environment variables on page C-4
- Hyperlinks in the Installer on page C-5
- Additional information for Solaris users only on page C-5
- Additional information for Linux users only on page C-6
- Repairing an installation on page C-8
- Uninstalling RealView Debugger on page C-8.

C.2.1 Types of installation

RealView Debugger provides four installation types:
- Minimum
- Typical
- Custom on page C-4
- Full on page C-4.

Minimum

For both Solaris and Linux users, this provides:
- RealView Debugger v1.6.1
- RealView Debugger v1.6.1 Utilities including Installer
- online help using HyperHelp v5.2.0.

Typical

For both Solaris and Linux users, this provides all the components of a Minimum installation plus:
- RealView Debugger v1.6.1 Examples
- full documentation suite in PDF and XML.

For Solaris users, this also provides:
- full documentation suite in DynaText
- DynaText v4.1.
Custom

Choose a Custom installation to select the RealView Debugger components that you want to install. Choose this to install DSP support if you want to use Multi-ICE direct connect.

Full

For both Solaris and Linux users, this provides all the components of both a Minimum installation and a Typical installation plus:

- RealView Debugger v1.6.1 Tools ADS
- RealView Debugger v1.6.1 Tools SDT.

C.2.2 Environment variables

When you install RealView Debugger, this sets the following environment variables:

**ARMROOT**  The root of all installed products from ARM. By default, this is `/opt/ARM` if you install RealView Debugger as root. If you install RealView Debugger as user, this is `~/ARM`.

**RVDEBUG_INSTALL**  The location of the platform-specific version of RealView Debugger, for example `install_directory/RVD/Core/1.6.1/61/solaris-sparc`.

**LD_LIBRARY_PATH**  The location of the `lib` directory for RealView Debugger, for example `install_directory/RVD/Core/1.6.1/61/solaris-sparc/lib`.

**PATH**  Includes RealView Debugger `bin` and `mwb` directories.

**HLPPATH**  The location of the RealView Debugger online help files, for example `install_directory/Documentation/RVD/1.6.1/release/unix/onlinehelp`.

**HHHOME**  The location of the RealView Debugger online help executable, for example `install_directory/Documentation/HyperHelp/5.2.0/release/unix`.

**DTEXT_PATH**  For Solaris users only. The location of the DynaText executable `dtext.exe`, for example `install_directory/Documentation/DynaText/4.1/release/unix/bin`. 
EBTRC  For Solaris users only. Used by DynaText to locate online documentation, for example

RVDEBUG_WEB_BROWSER  For Linux users only. This points to /usr/bin/htmlview.

The home directory
RealView Debugger requires a debugger-specific home directory to store user-specific settings such as your board file. Information about the other files that are stored in the debugger home directory is in an appendix to the RealView Debugger v1.6 Essentials Guide.

The location of this directory depends on the environment variables and command-line options defined when the debugger is started, for example if your $HOME environment variable is set to /home/user_name, the RealView Debugger home directory is /home/user_name/rvd.

C.2.3 Hyperlinks in the Installer
The RealView Debugger Installer displays the Release Notes containing hyperlinks to access the ARM Technical Support FAQs. These hyperlinks cannot be resolved by the Installer itself. Browse these files using the desktop link set up by the Installer or view the file readme.html at the top level on your installation CD.

C.2.4 Additional information for Solaris users only
Solaris users must be aware of the following information.

Setting environment variables
If you install RealView Debugger as user, the Installer writes to ~/.login. If this file does not exist, then the environment variables are lost and RealView Debugger does not run.

If you install RealView Debugger as user, the Installer also writes to ~/.profile and ~/.bash_profile. This means that you can use the bash shell as your default shell, rather than csh, if required.

The Installer reminds you to log out before starting RealView Debugger. It is not necessary to log out if you launch RealView Debugger from a desktop link on the Front Panel.
Setting up your license file

You must have a suitable license file to access all the features of RealView Debugger. This applies to all features and not just to separately licensed features such as multiprocessor debugging mode and Trace. When this file is installed, set the environment variable `ARMLMD_LICENSE_FILE` to point to this location.

**Note**

You must set this environment variable in your startup file to use the desktop link to launch RealView Debugger.

C.2.5 Additional information for Linux users only

Linux users must be aware of the following information.

**Installing under Gnome or KDE**

The RealView Debugger Installer sets up both Gnome and KDE desktop links. Therefore, you can install if you are running Gnome and then use KDE as your desktop. Similarly, you can install under KDE and then use Gnome as your desktop.

**Setting environment variables**

To install RealView Debugger on RedHat 7.2 or 7.3, you must set the `LANG` environment variable to either `en_GB` or `en_US`. If you do not do this, the Installer either hangs after you select the installation directory or it does not offer valid installation options.

If you install RealView Debugger as user, the Installer writes to `/etc/profile`. This means that you must run the bash shell to pick up these settings and launch RealView Debugger.

If you install RealView Debugger as user, the Installer also writes to both `~/.login` and `~/.bash_profile`. Ensure that these files exist and are set up correctly so that you can use RealView Debugger.

Be aware that:

- If these files do not exist, the Installer continues without warning. This means that the environment variables are lost and RealView Debugger does not run.
- If these files do exist, the Installer modifies them without warning. This preserves any environment variables you have set up already because the Installer only appends variables.
The Installer reminds you to log out before launching RealView Debugger. It is not necessary to log out if you launch RealView Debugger from a desktop link on the Front Panel.

**Setting desktop links**

If you install RealView Debugger as user, you must ensure that the following directories exist with write access to set up your desktop links:

**For Gnome**  
Ensure that you create  
/usr/share/gnome/apps/System_Wide_Application_Manager

**For KDE**  
Ensure that you create  
/usr/share/applnk/System_Wide_Application_Manager

To create these directories for Gnome, log on as root and then type:

```
$ mkdir /usr/share/gnome/apps/System_Wide_Application_Manager
$ chmod 777 /usr/share/gnome/apps/System_Wide_Application_Manager
```

If you are using the Gnome desktop manager on Linux RedHat 7.2 or 7.3, select **Programs → System Wide Application Manager** menu from the Front Panel to access the desktop links.

**Setting up your license file**

You must have a suitable license file to access all the features of RealView Debugger. This applies to all features and not just to separately licensed features such as multiprocessor debugging mode and Trace. When this file is installed, set the environment variable `ARMLMD_LICENSE_FILE` to point to this location.

--- **Note** ---

You must set this environment variable in your startup file to use the desktop link to launch RealView Debugger.

---

**XML Documentation**

The default XML browsers shipped with RedHat 7.3 do not display the XML documentation included with RealView Debugger v1.6.1. You must use Mozilla 1.0.1 or later to view these files.
C.2.6 Repairing an installation

RealView Debugger v1.6.1 does not support the option to repair any existing installation.

C.2.7 Uninstalling RealView Debugger

Before you start to uninstall RealView Debugger (or any of its components), ensure that there are no RealView Debugger executables running. Otherwise, this might result in the uninstaller leaving files on the system.

The file `install_directory/RV0/Tools/installer_touch` is created when you first install RealView Debugger and is used to track changes to the tools supported by RealView Debugger. If you uninstall RealView Debugger, this file is not removed and so you must remove it yourself.
C.3 Getting more information

This section describes how to get more information on RealView Debugger:

- Using DevZone
- Feedback on RealView Debugger.

C.3.1 Using DevZone

ARM provides a range of services to support developers using RealView Debugger. Among the downloads available are enhanced support for different hardware platforms through technical information and board description files. See http://www.arm.com to access these resources from ARM DevZone®, and for Release Notes, updates to documentation, and Frequently Asked Questions.

C.3.2 Feedback on RealView Debugger

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

1. Select Help → About... from the RealView Debugger main menu.
2. Click File an SPR on the About Box Information dialog box.
3. Complete all sections of the Software Problem Report.
4. 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.
5. Email the report to your supplier.
C.4 Changes to target configuration details

The major changes are the addition of Simulator Broker for ARMulator and access to RealView Network Broker to connect to remote workstations across a network.

This section describes additions to RealView Debugger v1.6 Target Configuration Guide:
- Using Simulator Broker connections to ARMulator
- Default configuration files on page C-13
- Target configuration entries on page C-14.

C.4.1 Using Simulator Broker connections to ARMulator

RealView ARMulator ISS is supplied with ARM debuggers and as a separate product. RealView ARMulator ISS runs on the same host computer as the debugger, and includes facilities for communicating with the debugger.

RealView ARMulator ISS is supplied with ARM debuggers and as a separate product. RealView ARMulator ISS runs on the same host computer as the debugger, and includes facilities for communicating with the debugger.

---

Note
RealView ARMulator ISS is not the same as the ADS 1.2 ARMulator described in the rest of this documentation suite for developers using RealView Debugger on Windows.

Note
RealView ARMulator ISS is an Instruction Set Simulator (ISS). It simulates the instruction sets and architecture of ARM processors, together with a memory system and peripherals. You can extend it to simulate other peripherals and custom memory systems.

Note
For details on using RealView ARMulator ISS, see the RealView ARMulator ISS v1.3 User Guide and Addendum 01 RealView ARMulator ISS v1.3 User Guide.

The Simulator Broker connection to RealView ARMulator ISS enables the debugger to connect to a target using a network connection. RealView Debugger must provide the Simulator Broker interface to RealView ARMulator ISS. ARMulator models communicate with Simulator Broker through an intermediate interface called SimRdi_Manager.

Note
RealView ARMulator ISS is the only way to connect to simulated ARM cores with RealView Debugger on Solaris or Linux.
Simulator Broker connections to RealView ARMulator ISS support the following features:

- instruction and data trace
- hardware breakpoints
- multiple instances of RealView ARMulator ISS.

To display the Connection Control window, either:

- Click on the hyperlink in the File Editor pane, if available.
- Select **File → Connection → Connect to Target...** from the default Code window.

To connect to RealView ARMulator ISS, choose **Start ARMulator simulator** under the **localhost Simulator Broker** entry, shown in Figure C-1.

---

**Note**

If you are licensed to use RealView Debugger extensions, the Connection Control window includes a **Connect** tab and a **Synch** tab. In single processor debugging mode these are not visible.

---

To configure your RealView ARMulator ISS ARMulator connections, right-click anywhere along the line to display the context menu in the usual way, shown in Figure C-2 on page C-12.
Figure C-2 Configuring ARMulator connections

For full details on setting up your ARMulator connections, see the chapter describing configuring custom connections in *RealView Debugger v1.6 Target Configuration Guide*. However, be aware of the following changes when using this dialog box:

- Some cores are not available when you are connected using RealView ARMulator ISS, for example StrongARM, ARM8 and the simulated ETM functionality.
- Use the Additional Modules control group to define *Floating Point Accelerator* (FPA) and *Vector Floating Point* (VFP) options (VFPv1 is no longer available).
- If you choose **Real-time**, do not enter a value for Speed.
- If you choose **Emulated**, specify clock speed in Hz, for example 50000Hz.
- RealView Debugger cannot make use of Memory Map Files when making ARMulator connections using RealView ARMulator ISS.

RealView ARMulator ISS enables you to make multiple connections, shown in Figure C-3 on page C-13.
Figure C-3 Multiple connections to ARMulator in the Connection Control window

Figure C-3 shows three instances of RealView ARMulator ISS. The current connection appears in the title bar of the default Code window:

rvdebug - RVDEBUG = @SimARM_3:Sim [Unattached]

For full details on using the Connection Control window and configuring targets in RealView Debugger, see *RealView Debugger v1.6 Target Configuration Guide*.

---

**Note**

When you are working with multiple connections, you might find that disconnecting does not remove the entry from the Connection Control window immediately. If you try to reconnect using the same entry, this fails.

C.4.2 Default configuration files

Because you are using Simulator Broker connections to RealView ARMulator ISS, RDI configuration files, described in the documentation suite, are not installed when running RealView Debugger on Solaris or Linux, for example *.rbe or *.cnf files. Other configuration files are installed in

*install_directory/RVD/Core/1.6.1/61/solaris-sparc/etc:*

- board file, for example `rvdebug.brd`
- JTAG files, for example `arm.jtg`
- Board/Chip definition files, for example `CM940T.bcd`

For full details on these files and the settings they contain, see the chapter describing configuring custom targets in *RealView Debugger v1.6 Target Configuration Guide.*
C.4.3 Target configuration entries

RealView ARMulator ISS enables you to connect to remote simulators and On-Chip Debugging (OCD) based emulators such as Multi-ICE direct connect.

This means that target configuration entries related to remote connections are now supported by RealView Debugger, for example the Remote group, shown in Figure C-4.

![Figure C-4 Remote entries in the Connection Properties window](image)

See Connecting to remote hosts on page C-24 for details on how to make remote connections using these settings.

Using Trace

Developers working on Solaris or Linux can use Trace with RealView ARMulator ISS.

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. These settings are not used with RealView ARMulator ISS.

Note

RealView Debugger Trace support is a separately licensed component. See the chapter describing tracing in RealView Debugger v1.6 Extensions User Guide for full details on using this extension.
Using the DSP

The DSP support in RealView Debugger is invoked by connecting the debugger to an Oak or TeakLite processor. Developers working on Solaris or Linux can connect to target hardware using Multi-ICE direct connect. See Connecting to remote hosts on page C-24 for details.

Note

RealView Debugger DSP support is a separately licensed component. See the chapter describing DSP support in RealView Debugger v1.6 Extensions User Guide for full details on using this extension.
C.5 Changes to GUI and general user information

This section describes:

- Getting started with RealView Debugger
- Making connections using the CLI on page C-18
- Debug views on page C-18
- Using breakpoints on page C-19
- Global configuration options on page C-19
- Column resizing on page C-20
- New Help menu item on page C-22
- Changes to the desktop on page C-22
- RealView Debugger examples on page C-23.

C.5.1 Getting started with RealView Debugger

The RealView Debugger Control Panel, RVDEBUGCP, provides access to the debugger for developers using Solaris and Linux. Start the debugger to see this window, shown in Figure C-5.

By default, launching the Control Panel starts RealView Debugger without a splash screen.

Click on the required icon to launch:

**rvdebug**
- Access startup options for RealView Debugger.
- Double-click on this icon to start the debugger in default mode and display the Code window.

**Editor**
- Access startup options for mwedit, the standalone editor installed as part of RealView Debugger. Use this to edit source files and work on project files.
- Double-click on this icon to start mwedit in default mode and display the standalone editor window.

![RealView Debugger Control Panel](image)
Running RealView Debugger from the Control Panel

Left-click on the **rvdebug** icon to specify arguments when using the command-line method of starting RealView Debugger:

**Start With Arguments...**

Displays a dialog box where you can specify startup options, for example to specify a workspace to use in this session or to write to a log file.

**Set Default Arguments...**

Displays the Tools Attributes dialog box where you can specify the start directory and arguments for running RealView Debugger. If you set these, they are used each time the debugger runs from the Control Panel.

See Chapter 1 *Starting to use RealView Debugger* for details of startup options and command-line arguments.

**Note**

If you are using Solaris or Linux, you can use the `-cmd` argument to start the command-line debugger, for example `rvdebug -cmd`.

Running an editor from the Control Panel

Left-click on the **Editor** icon to specify arguments when starting mwedit:

**Start With Arguments...**

Displays a dialog box where you can specify startup options, for example to start the editor with a file loaded.

**Set Default Arguments...**

Displays the Tools Attributes dialog box where you can specify the start directory and arguments for running mwedit. If you set these, they are used each time the standalone editor runs from the Control Panel.

**Note**

If you are using mwedit in standalone mode, certain menu options are not available. For example, you cannot access workspace options, connection details, or project-related operations from the mwedit menu bar. These options are available if you start the standalone editor from within the Code window.

See Chapter 12 *Editing Source Code* for details on using editors in RealView Debugger.
C.5.2 Making connections using the CLI

You can use the CLI CONNECT command to make numbered connections as described in the documentation suite, for example:

connect,route 1
cconnect 6

where the connection id is used to identify the target.

You can connect to remote connections in the same way.

Making named connections

You can also make named connections, for example:

connect @new_ARM
connect @new_OAK@Remote_Debug
connect @new_ARM@Remote_MICE

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

connect @new_ARM@localhost
connect @new_ARM@Remote_ARM_Debug

See the description of the CONNECT command in RealView Debugger v1.6 Command Line Reference Guide for details on connecting to targets this way.

C.5.3 Debug views

Be aware of the following when examining registers, memory contents, and variables:

- When you are using RealView ARMulator ISS software simulator to simulate an ARM-based debug target, you must load an image, or write to the PC, to begin execution.

- Loading an image with RealView ARMulator ISS does not automatically send a reset. To reset at the same time as an image load, send a reset command before you load, or reload, the image.

- If you are using semihosting with RealView ARMulator ISS, you cannot use the Stop button during the semihosting input.
The Register pane contains the following tabs when you are using RealView ARMulator ISS:

- **Core**, showing the base registers for the connected target processor.
- **CycleCount**, showing internal debugger variables.

Add core-specific registers to the Register pane using custom tabs. See *Addendum 01 RealView ARMulator ISS v1.3 User Guide* for details.

When you are using RealView ARMulator ISS, registers in some ARM cores are incompletely modeled, that is:

- ARM925T, wait for interrupt
- ARM966E-Sr2, TCM register size missing
- ARM946E-Sr1, r13 trace PID
- ARM926EJ-S, r13 context ID writing to the wrong register
- ARM720Tr4, does not distinguish trace PID and FCSE r13.

### C.5.4 Using breakpoints

Be aware of the following when working with breakpoints:

- When you are using RealView ARMulator ISS software simulator to simulate an ARM-based debug target, watchpoints are available. These are called hardware breakpoints in RealView Debugger and can be accessed through the *Debug* menu. These are implemented using a memory hook.

- If you are using RealView ARMulator ISS, hardware breakpoints can use address ranges, data values, and data value range tests. They can also include size tests, mode tests, and pass counts.

- If you are using RealView ARMulator ISS, hardware breakpoints can be chained to form complex tests.

- If you are using RealView ARMulator ISS, a reset command does not clear breakpoints or tracepoints.

For full details, see Chapter 4 *Working with Breakpoints*.

### C.5.5 Global configuration options

The first time you run RealView Debugger after installation, it creates a default workspace to define your initial working environment. Two files are created in your RealView Debugger home directory to store settings:

- **rvdebug.aws** Contains workspace-specific settings that apply to the current workspace.
rvdebug.def  Contains global configuration options that apply to all workspaces, or are used when working without a workspace.

The rvdebug.def file replaces the rvdebug.ini file described in the RealView Debugger documentation suite. Select Tools → Options... to make changes to settings in this file.

For full details on how these files are used, see Chapter 10 Configuring Workspace Settings.

Source coloring

The standard C/C++ source coloring is auto-enabled based on file extension. Use the File_extensions setting in the Source_coloring group to specify a comma-separated list of file extensions that, when loaded, trigger source code coloring.

In the current release on Solaris and Linux, any changes made to the default list of valid extensions are ignored by RealView Debugger.

For full details on these settings, see Text on page A-9.

C.5.6 Column resizing

When using the Code window, the following panes use two columns to display debug data:

•     Call Stack
•     Break/Tracepoints
•     Watch
•     Process Control.

To make the display easier to read, you can change the size of the first column using a new menu option available only on Solaris and Linux:

1. Select File → Reload Image to Target to reload the image dhrystone.axf.
2. Click on the Src tab to view the source file dhry_1.c.
3. Set a simple breakpoint by double-clicking on line 150.
4. Click Go to start execution.
5. Enter 5000 when asked for the number of runs.

The program starts and then stops when execution reaches the breakpoint at line 150. The red box marks the location of the Program Counter (PC) when execution stops.

6. Select View → Pane Views → Call Stack to view the Call Stack pane.
7. Right-click somewhere in the header columns to display the context menu, shown in Figure C-6.

![Figure C-6 Column resizing in the Watch pane](image)

8. Select the option **Set Width Of Column 1** to display the prompt box where you can specify the size you want, shown in Figure C-7.

![Figure C-7 Column resizing prompt box](image)

When the prompt box first appears, it contains the current setting. Enter a value between 1 and 128.

9. Click **Set** to confirm the new setting. Click **Cancel** to leave the size unchanged.

If you change the size of a column, it holds for all windows in the current session. Default sizes are restored at each start up.

You cannot change the size of columns in the following panes:
- Register
- Stack
- Symbol Browser
- Memory.

If you resize columns be aware of the following:
- You can only resize the first column in any two-column display.
- Columns can only be adjusted to within one monospaced character position.
- The second column is automatically set to accommodate the longest item.
- Changing the column size applies to all tabs in a multitab display, that is the Watch, Call Stack, and Process Control panes.
C.5.7 New Help menu item

ARM provides a range of services to support developers using RealView Debugger. Among the downloads available are enhanced support for different hardware platforms through technical information and board description files. The Help menu contains a new option to give access to the new Updates and Utilities area. See http://www.arm.com to access these resources from ARM DevZone.

C.5.8 Changes to the desktop

This section summarizes differences when using RealView Debugger desktop on Solaris or Linux.

Code window

Be aware of the following when working in the default Code window:

- Code windows are identified by a title bar. This changes as you open new windows, for example RVDEBUG, RVDEBUG_1, RVDEBUG_2.
- The Code window title bar identifies the vehicle you are using to make the connection and the connection number. The target processor is not shown.
- Code windows do not have a Color Box to identify target connections.
- Other windows, such as the Resource Viewer, do not include a Color Box to identify the calling Code window.
- Code window panes cannot float.
- Tooltips are not available if you hover over a toolbar button.
- The editing control called Tooltip Evaluation that provides hover-style evaluation in different code views is not available.
- The Code window main menu includes a Help menu. This is located at the right of the default window.
- Select Help → About... from the RealView Debugger main menu to display the About Box Information dialog where you can submit a Software Problem Report.
- There are no status display areas at the bottom of the Code window.
Pane controls

Panes do not include title bars to describe their content. However, each pane contains
the controls:

Pane Content

Click this button to display the Pane Content menu where you can
change the debug view in the pane.

The selected option in the menu indicates the current view.

The visual controls are at the bottom of the Pane Content menu. Use
these to hide the pane. The option to float a pane is not available.

Pane Menu

Click this button to display the Pane menu.

Use this to:

• change the display format
• change how pane contents are updated
• extract data from the pane.

The options available from this menu depend on the pane.

Expand/Collapse Pane

Use the pane slide controls to change the size of a pane.

You cannot switch the Side pane from its location to the right of the File Editor pane.

C.5.9 RealView Debugger examples

If you choose a Typical or Full installation, the RealView Debugger examples are
installed in install_directory/RVD/Examples. The dhrystone example project, and the
executable built by this project, are used in the documentation suite.

Note

The -D compiler switch is set to TIME to control how timing measurements are made on
Solaris and Linux platforms.

See the tutorial in RealView Debugger v1.6 Essentials Guide for more details on
building this example project.
C.6 Connecting to remote hosts

Execution vehicles can reside on the same workstation as RealView Debugger or any other workstation on your network. These services are handled by RealView Connection Broker, rvbroker.exe. This section describes how RealView Debugger handles remote connections:

- **RealView Network Broker**
- Starting RealView Network Broker on remote hosts on page C-25
- Accessing remote hosts on page C-25
- Specifying the remote host on page C-25
- Connecting to a remote host simulator on page C-26
- Connecting to a remote host emulator on page C-28
- Using a hosts file on page C-31
- Disconnecting remote connections on page C-31.

C.6.1 RealView Network Broker

RealView Connection Broker operates in two modes:

**Local**

Operating as RealView Connection Broker, this runs on your local workstation and enables you to access targets on the local workstation.

**Remote**

Operating as RealView Network Broker, this runs on a remote workstation and makes specified targets on that workstation available to other workstations connected to the same network.

Local host simulators are available immediately from the Connection Control window. If you expand the Simulator Broker entry, ready to connect to a simulator, RealView Debugger starts RealView Connection Broker in local mode to manage your connection.

RealView Debugger v1.6.1 supports RealView Connection Broker on both Windows and UNIX platforms. This means that it is possible to connect to a remote Windows workstation running ARMulator.
C.6.2 Starting RealView Network Broker on remote hosts

Any remote workstation that is to give access to simulators or emulators must be running RealView Connection Broker in remote mode, that is RealView Network Broker. This makes the workstation visible to other users across the network. You can start RealView Network Broker on a remote host in different ways:

- If the remote workstation is running UNIX and the `rsh` command is available at the local workstation, the local workstation can start RealView Network Broker on the remote workstation.

- If the remote workstation is running UNIX and the `rsh` command is not available locally, RealView Network Broker must be started explicitly on the remote UNIX workstation, for example:

  ```
  install_directory/RV0/Core/1.6.1/61/solaris-sparc/bin/rvbroker 0 remote
  ```

  You can set this up in a `.login` file or in the startup group.

- If the remote workstation is running Windows, RealView Network Broker must be started explicitly on that workstation. For example, select Start → Programs → ARM RealView Debugger v1.6.1 from the Windows Start menu and then select RealView Network Broker.

For full details on how to start RealView Network Broker on a remote workstation see Chapter 1 Starting to use RealView Debugger.

C.6.3 Accessing remote hosts

RealView Network Broker enables you to connect to remote simulators and OCD-based emulators such as Multi-ICE direct connect.

To access remote connections, debug target configuration files are held on the local workstation and pushed across the network connection to the remote host. RealView Network Broker must be running on the remote workstation to provide access to the local simulators and emulators.

C.6.4 Specifying the remote host

To access a remote host simulator or emulator using RealView Network Broker you must define the location of the remote workstation in your target configuration settings saved in your local `.brd` file. There are two ways to do this:

**RVBROKER**

This entry specifies the location of a host that supports simulators. The Connection Control window uses this entry to define remote targets. See Connecting to a remote host simulator on page C-26 for details.
rvbroker.brd

This entry enables the inclusion of a special hosts file containing a list of all network nodes that can run, or are running, RealView Network Broker. See Using a hosts file on page C-31 for details.

You can combine both types of entry in the same configuration file.

C.6.5 Connecting to a remote host simulator

To configure your debug environment to access a remote host simulator:

1. Start the remote connection broker, that is RealView Network Broker.
   RealView Debugger does not have to be running on the remote workstation. Only RealView Network Broker is required to make the simulators and emulators visible across the network.

2. Log onto the local workstation and start RealView Debugger without connecting to a target.

3. Select File → Connection → Connection Properties... to display the Connection Properties window.

4. Configure the debug target environment on the local workstation:
   a. Copy the existing RVBROKER=localhost entry to create a duplicate RVBROKER group to define the remote host, for example RVBROKER=Remote_Debug.
   b. Edit the Remote settings values page in the new group to enter the hostname of the remote workstation, for example Hostname=armpc41, shown in Figure C-8 on page C-27. Specify the full domain name where necessary, for example Hostname=armpc41.ournet.arm.com.
You can use the IP address, for example Hostname=192.168.2.212.
It is not necessary to specify the host name of the remote workstation if the RVBROKER group name is also the host name of the remote workstation on your network, for example RVBROKER=PC44. Where the Hostname entry is filled in, this is used and the group name is irrelevant.

c. Edit the Description settings value in the new group to identify the remote connection, for example Remote sim on network test pc.

5. Select File → Save and Close to save the new settings and close the Connection Properties window.

6. Display the Connection Control window to see the new remote connection, shown in Figure C-9.

7. Expand the new Remote_Debug entry to view the available simulators made visible by the remote connection broker.
This entry is empty if there are no remote simulators available on the specified workstation.

8. Click on an entry to make the remote connection.

You can create multiple RVBROKER groups if required. Use unique names to identify the different remote hosts.

Ending a debugging session on the remote workstation, and closing down RealView Debugger, does not terminate the remote connection broker. RealView Network Broker must be shut down explicitly when it is no longer required.

C.6.6 Connecting to a remote host emulator

RealView Debugger enables you to connect to OCD-based emulators running on remote hosts across a network. To do this you must create a new CONNECTION entry in your board file and use this to specify the hostname of the remote workstation.

To configure your debug environment to access a remote host emulator:

1. Log onto the remote workstation and start RealView Debugger without connecting to a target.

2. Select File → Connection → Connection Properties... to display the Connection Properties window.

3. In the Connection Properties window, expand the connection that you are using:
   a. (*.rbe) ARM RDI Configuration Entries
   b. ...\multiice.rbe

4. Click on CONNECTION=Multi-ICE so that it is highlighted with a red border. The right pane displays a set of properties. Configure the following entries:
   - Expand Connect_with and set Manufacturer=ARM-A-RR.
   - Set Configuration=multiice.cnf.
   - Set Shared=True.
   - Set the BoardChip_name if you are using a specific .bcd file. Link other definitions as required to the CONNECTION.

5. Save the changes to the .brd file.

6. Start the remote connection broker, that is RealView Network Broker. RealView Debugger does not have to be running on the remote workstation. Only RealView Network Broker is required to make the shared configuration details visible across the network.
Ensure that Multi-ICE server is not running.

7. Log onto the local workstation and start RealView Debugger without connecting to a target.

8. Select **File → Connection → Connection Properties...** to display the Connection Properties window.

9. Right-click on the entry .../rvdebug.brd in the left pane.

10. Select **Make New Group...** to display the Group Type/Name selector dialog.

11. Select the type of group you want to use, that is **CONNECTION**.

12. In the Group Name data field change the name from *new* to something suitable for your target, using only alphanumeric characters, underscore _, and dash -. This example shows **Remote_MICE**.

13. Click **OK** to create the new group, shown in Figure C-10.

   ![Figure C-10 Viewing the new connection](image)

14. Edit the Connect_with settings values page to specify the connection.

   Right-click on the **Manufacturer** entry and select **Edit as String** from the context menu. Use in-place editing to set this to **ARM-ARM-PP**.

15. Right-click on the **Configuration** entry and use in-place editing to set this to the required JTAG file. If, for example, you are connecting to a single ARM core, set this to `install_directory/RVD/Core/1.6.1/61/solaris-sparc/etc/arm.jtg`.

16. Right-click on the **BoardChip_name** entry, to specify a .bcd file to use. For example, select **AP** to use an ARM Integrator/AP. Link other definitions as required to the new **CONNECTION**.
17. Edit the Remote settings values page in the new group to enter the hostname or IP address of the remote workstation, shown in Figure C-11.

![Figure C-11 Configuring a remote debug target](image)

18. Select File → Save and Close to save the new settings and close the Connection Properties window.

19. Display the Connection Control window to see the new remote connection, shown in Figure C-12.

![Figure C-12 Remote target in the Connection Control window](image)

20. Expand the new Remote_MICE entry to view the available emulators made visible by the remote connection broker.
   This entry is empty if there are no remote emulators available on the specified workstation.

21. Click on an entry to make the remote connection.
C.6.7 Using a hosts file

The `rvbroker.brd` entry in the configuration file, shown in Figure C-8 on page C-27, enables the inclusion of a special hosts file containing a list of all network nodes that can run, or are running, RealView Connection Broker. This enables multiple users to access simulators, emulators, and Embedded Virtual Machines (EVMs) anywhere on the network by sharing this single file.

If you are using a hosts file, this is the same as adding `RVBROKER=` entries to your board file to manage remote connections across your network. However, the connection details are contained in the `rvbroker.brd` file and not in the `rvdebug.brd` file.

When RealView Debugger reads a board file, for example `rvdebug.brd`, it searches for an `rvbroker.brd` file to use. The debugger searches:

1. in `install_directory/RVD/Core/1.6.1/61/solaris-sparc/etc`
2. your home directory
3. the current working directory.

Where you are using a local hosts file, it must be installed in your home directory. Install the hosts file in `install_directory/RVD/Core/1.6.1/61/solaris-sparc/etc` for shared access. The search order defines the precedence when using multiple files in different locations.

C.6.8 Disconnecting remote connections

You can use the CLI command `DISCONNECT` to disconnect from a remote connection using either the connection id or the target name, for example:

```
disconnect 9
disconnect @RSimARM_12
```

where the name `RSimARM_12` identifies the remote target in the Connection Control window.

See the description of the `DISCONNECT` command in `RealView Debugger v1.6 Command Line Reference Guide` for details on disconnecting this way.
Glossary

The items in this glossary are listed in alphabetical order, with any symbols and numerics appearing at the end.

**Access-provider connection**
A debug target connection item that can connect to one or more target processors. The term is normally used when describing the RealView Debugger Connection Control window.

**Address breakpoint**
A type of breakpoint.

*See also* Breakpoint.

**ADS**
*See* ARM Developer Suite.

**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 is not available.

**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.
### Glossary

<table>
<thead>
<tr>
<th><strong>ARM state</strong></th>
<th>A processor that is executing ARM (32-bit) instructions is operating in ARM state. See also Thumb state</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARMulator</strong></td>
<td>ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.</td>
</tr>
<tr>
<td><strong>Asynchronous execution</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>ATPCS</strong></td>
<td>ARM-Thumb Procedure Call Standard.</td>
</tr>
<tr>
<td><strong>Backtracing</strong></td>
<td>See Stack Traceback.</td>
</tr>
<tr>
<td><strong>Big-endian</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Board</strong></td>
<td>RealView Debugger uses the term <em>board</em> to refer to a target processor, memory, peripherals, and debugger connection method.</td>
</tr>
<tr>
<td><strong>Board file</strong></td>
<td>The <em>board file</em> is the top-level configuration file, normally called <em>rvdebug.brd</em>, that references one or more other files.</td>
</tr>
<tr>
<td><strong>Breakpoint</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Conditional breakpoint</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Context menu</strong></td>
<td>See Pop-up menu.</td>
</tr>
<tr>
<td><strong>Core module</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>CPSR</strong></td>
<td>Current Program Status Register. See also Program Status Register.</td>
</tr>
<tr>
<td><strong>DCC</strong></td>
<td>See Debug Communications Channel.</td>
</tr>
</tbody>
</table>
Debug Communications Channel (DCC)
A debug communications channel enables data to be passed between RealView Debugger and the EmbeddedICE logic on the target using the JTAG interface, without stopping the program flow or entering debug state.

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

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.

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.
**Glossary**

**Floating Point Emulator (FPE)**
Software that emulates the action of a hardware unit dedicated to performing arithmetic operations on floating-point values.

**FPE**
See Floating Point Emulator.

**Halfword**
A 16-bit unit of information.

**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 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 focused on silicon chip testing methods. Many debug and programming tools use a Joint Test Action Group (JTAG) interface port to communicate with processors. For further information refer to IEEE Standard, Test Access Port and Boundary-Scan Architecture specification 1149.1 (JTAG).

**JTAG**
See Joint Test Action Group.

**JTAG interface unit**
A protocol converter that converts low-level commands from RealView Debugger into JTAG signals to the processor, for example to the EmbeddedICE logic and the ETM.

**Little-endian**
Memory organization where the least significant byte of a word is at the lowest address and the most significant byte is at the highest address of the word.

See also Big-endian.

**Multi-ICE**
A JTAG-based tool for debugging embedded systems.

**Pop-up menu**
Also known as Context menu. A menu that is displayed temporarily, offering items relevant to your current situation. Obtainable in most RealView Debugger windows or panes by right-clicking with the mouse pointer inside the window. In some windows the pop-up menu can vary according to the line the mouse pointer is on and the tabbed page that is currently selected.
**Processor core**
The part of a microprocessor that reads instructions from memory and executes them, including the instruction fetch unit, arithmetic and logic unit and the register bank. It excludes optional coprocessors, caches, and the memory management unit.

**Profiling**
Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code.

**Program Status Register (PSR)**
Contains information about the current execution context. It is also referred to as the *Current PSR* (CPSR), to emphasize the distinction between it and the *Saved PSR* (SPSR), which records information about an alternate processor mode.

**PSR**
See Program Status Register.

**RDI**
See Remote Debug Interface.

**RealView Compilation Tools**
*RealView Compilation Tools* is a suite of tools, together with supporting documentation and examples, that enables you to write and build applications for the ARM family of RISC processors.

**RealView Debugger Trace**
A software product add-on to RealView Debugger that extends the debugging capability with the addition of real-time program and data tracing.

**Remote Debug Interface (RDI)**
The *Remote Debug Interface* (RDI) is an ARM standard procedural interface between a debugger and the debug agent. RDI gives the debugger a uniform way to communicate with:

- a simulator running on the host (for example, 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).

**Remote_A**
Remote_A is a software protocol converter and configuration interface. It converts between the RDI 1.5 software interface of a debugger and the Angel Debug Protocol used by Angel targets. It can communicate over a serial or Ethernet interface.

**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.

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.

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.
### Glossary

**Target**

The target hardware, 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.

**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*

**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.

**Tracing**

The real-time recording of processor activity (including instructions and data accesses) that occurs during program execution. Trace information can be stored either in a trace buffer of a processor, or in an external trace hardware unit. Captured trace information is returned to the Analysis window in RealView Debugger where it can be analyzed to help identify a defect in program code.

**Trigger**

In the context of breakpoints, a trigger is the action of noticing that the breakpoint has been reached by the target and that any associated conditions are met.

In the context of tracing, a trigger is an event that instructs the debugger to stop collecting trace and display the trace information around the trigger position, without halting the processor. The exact information that is displayed depends on the position of the trigger within the buffer.

**TVS**

*See Target Vehicle Server.*

**Vector Floating Point (VFP)**

A standard for floating-point coprocessors where several data values can be processed by a single instruction.

**VFP**

*See Vector Floating Point.*

**Watch**

A watch is a variable or expression that you require the debugger to display at every step or breakpoint so that you can see how its value changes. The Watch pane is part of the RealView Debugger Code window that displays the watches you have defined.

**Watchpoint**

In RealView Debugger, this is a hardware breakpoint.
**Word**  
A 32-bit unit of information.
Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

About this book viii
Active
  build target configuration 11-74
  project 11-12
Add/Copy/Edit Memory Map
dialog 5-5
Add/Edit Macros
dialog 9-7
ASSEMBLE group B-46
  Assembly B-55
  Listings B-52
  Messages B-54
  Preprocessor B-51
  Sources B-49
Audience, intended viii
Autobinding
  see Projects
Auto-projects
  changing settings 2-13
  closing 2-14
  creating for images 2-11
displaying settings 2-12
  in the Process Control pane 2-11,
  11-85
  merging 11-29, 11-36
  merging options 11-37
  merging SETTINGS group 11-37
  overview 11-5
  Project Properties groups 11-49
  properties 2-11, 2-12
  saving settings 2-14
  see also Projects
Autoscope 3-3
  in assembly language 3-3
  red box 3-3

B

Board file 2-2
Book, about this viii
Breakpoint units 4-8
  in Break/Tracepoints pane 4-8
Breakpoints 4-2
  Access 4-35
  actions 4-18
  attaching macros 4-21
  breakpoint types 4-2
  breakpoint units 4-8
  clearing 4-39
  clearing all 4-16, 4-40
  clearing quickly 4-8
  complex 4-2, 4-30
  Complex Breakpoints menu 4-30
  conditional breakpoints 4-13, 4-18
  copying in the Break/Tracepoints
    pane 4-37
  creating a favorite 4-41
  Debug menu mappings 4-37
  disable at cursor 4-16
  disabling 4-39
  editing in the Break/Tracepoints
    pane 4-37
  enable at cursor 4-16
  EXEC 4-35
  hardware 4-3, 4-6, 4-24
hardware support in a DSP  4-27, 4-29
in projects  11-66
in the Break/Tracepoints pane  4-33, 4-35
INSTR  4-35
managing actions  4-19
managing qualifiers  4-18
markers  4-7, 4-35
named  4-15, 11-66
operations  4-16
Processor Events  4-15
Qualifiers  4-18
Read  4-35
red disc  4-5, 4-7
saving as favorites  4-41, 4-42
setting hardware breakpoints  4-24
setting hardware breakpoints using a DSP  4-29
setting hardware tests  4-26
setting in disassembly-level view  4-6
setting in other ways  4-9
setting in source-level view  4-5
setting in the Memory pane  4-9
setting in the Stack pane  4-9
setting on a branch  4-7
setting on a symbol  4-5
setting on an instruction  4-7
setting on inlined functions  4-8
setting on multiple statements  4-8
setting on the statement  4-5
setting simple breakpoints  4-13
setting using drag-and-drop  4-9
simple  4-2, 4-10
Simple Breakpoints menu  4-10
software  4-3, 4-6
toggle at cursor  4-16
using a mask  4-27
using a range  4-27
using with macros  9-5
viewing actions  4-20
viewing in disassembly-level view  4-7
viewing in source-level view  4-7
viewing qualifiers  4-20
white disc  4-8
Write  4-35
yellow disc  4-8
Browsers
accessing  8-2, 8-17
applying a filter  8-13
browsing C++ classes  8-2, 8-15
browsing functions  8-6
browsing modules and files  8-3
browsing variables  8-10
filter metacharacters  8-13
Function List  8-2, 8-6
in RealView debugger  8-2
Module/File List  8-2, 8-3
Register List  8-2
scope  8-2
using filters  8-13
Variable List  8-2, 8-10, 8-15
Build
dialog  11-43
Build errors
finding  11-16, 11-45
BUILD group  B-60
Link_Advanced  B-66
Listings  B-64, B-75
Messages  B-65, B-76
Pre_Post_Lib  B-77
Pre_Post_Link  B-70
RVDEBUG_Commands  B-71, B-79
Symbol_Control  B-68
Build model
in projects  11-41, 11-59
see also  Build target configurations
Build target configurations
active  11-74
assigning a setting to  11-76
assigning multiple settings to  11-79
base settings for  11-73
creating  11-74
default  11-58, 11-72
deleting  11-74
groups  11-73
in projects  11-8
removing a setting from  11-77
Build tools  11-7, 11-18
defining  11-19
genmake.loc  11-19
see also  Projects
Build-Tool Properties window  11-19
BUILD_LIB group  B-72
Complex breakpoints  
see Breakpoints

Conditional breakpoints  
see Breakpoints

CONFIGURATION group  B-22

Connecting  2-2

Connection Broker  1-6
  local host connection  1-6
  remote host connection  1-6
  shut down in remote mode  1-7

Container project  11-4
  creating  11-34
  properties groups  11-49
  see also  Projects
  specifying properties  B-8

Context
  controls  6-26
  see Scope

Create Container Project
  dialog  11-34

Create Custom Project
  dialog  11-32

Create Library Project
  dialog  11-31

Create New Project
  dialog  11-29

Create Standard Project
  dialog  11-31

CUSTOM group  B-58

Custom project  11-4
  creating  11-32
  properties groups  11-49
  see also  Projects

Customizing
  registers  6-11

C++
  browsing classes  8-15
  classes in the Symbol Browser  8-15
  finding a class definition  8-16
  viewing classes  8-16

D

Data values
  saving as favorites  6-40

DCC  6-2

DCC semihosting  6-2, 6-3, 6-11

Debug communications channel  6-2

Debug menu  3-9

Debug target
  configuration  2-2
  connecting  2-2
  Connection Broker  1-6
  memory mapping  5-2, 5-7, 5-8
  resetting processor  2-19, 3-11

Debugger
  defining in workspace  10-14
  internals  6-9

Dependencies
  see Projects

Dialog box
  Add/Copy/Edit Memory Map  5-5
  Add/Edit Macros  9-7
  Build  11-43
  Create Container Project  11-34
  Create Custom Project  11-32
  Create Library Project  11-31
  Create New Project  11-29
  Create Standard Project  11-31
  Fill Memory with Pattern  7-18
  Find  13-15
  Find in Files  8-16, 13-7
  Flash Memory Control  7-9, 7-10
  Function List  8-6
  HW Break if in Range  4-30
  HW Break if X, then if Y  4-31
  HW Break on Data Value match  4-32
  HW While in func/range, Break if X  4-31
  Insert Template  12-21
  Interactive Memory Setting  7-12
  Interactive Register Setting  7-14
  Jump to Function  13-10
  Load File to Target  2-3
  Module/File List  8-3
  New/Edit Favorite  4-41
  Pair Matching  13-12
  Project Control  11-81
  Save Multiple  12-9

Select File to Include Commands
  from  3-15

Select File to Journal to
  3-14

Select File to log the STDIO to
  3-14

Select File to Log to
  3-14

Select Project to Copy from
  11-35

Select Project to Open
  11-21

Set Address/Data Break/Tracepoint
  4-10

Settings:
  List Manager  11-63, 11-64

Simple Break if X  4-13

Simple Break if X, N times  4-14

Simple Break if X, when Y is True  4-14

Source Search Paths  3-16

Upgrade Project ToolChain  11-25

Upload/Download file from/to
  Memory  7-15

Variable List  8-10

Directories
  home  1-8
  installation  1-8
  used in RealView Debugger  1-8

Dragging and dropping text  12-13

EDIT

Editor button
  in Editor Buffer List  12-7

Edit settings
  in the workspace  12-17

Editing Controls
  auto indent  12-16
  in File Editor  12-15
  persistence  12-16
  setting in workspace  12-17
  shift width  12-16
  tab size  12-16
  text color  12-16
  tooltip evaluation  12-16
  vi mode  12-16

EDITOR  12-20

Editor
  see File Editor

Editor Buffer List  12-6
  displaying files  12-7
  Edit button  12-7
  flushing the buffer  12-6
Index

in File Editor 12-5
Minimize button 12-7
ReScan button 12-7
Restore button 12-7
Save button 12-7
EDIT_CMD 12-20
Environment variables
ARMLIB 11-64
EDITOR 12-20
EDIT_CMD 12-20
JOULOG_UNBUF 3-13
RVDEBUG 3-16
RVDEBUG_HOME 1-5
RVDEBUG_INSTALL 1-4
RVDEBUG_SHARE 1-5
VISUAL 12-20
Execution controls 3-6
from the Debug menu 3-10
submitting commands 3-2
using shortcuts 3-8

F
Favorites
breakpoint 4-41
data value 6-40
watch 6-38
Feedback xv
File Editor
changing case of text 12-11, 12-12
configuring 12-4
creating templates 12-22
current workspace 12-17
dirty file marker 12-3
editing templates 12-22
Editor Buffer List 12-5
ex commands 12-26
features 12-2
File Editor pane 12-2
File field 12-3
file tabs 12-3
flushing the buffer 12-6
inserting templates 12-21
managing your session 12-15
moving files between windows 12-14
personal editor 12-20
reverting to previous version 12-13
search controls 13-15
searching text 13-15
shifting text 12-12
Source control button 12-3
standalone editors 12-5, 12-19
template examples 12-23
text coloring 12-12
text formatting 12-11
using multiple files 12-4
Files
adding to projects 11-53, 11-62, 11-64, 11-65
binary 12-9
board file 2-2
closing in File Editor 12-9
close format 12-9
close to DOS 12-9
close to UNIX 12-9
excluding from build 11-55
including from the Debug menu 3-15
journal 3-13
log 3-13
opening in File Editor 12-9
re-including in a build 11-56
removing from projects 11-54
saving multiple 12-9
STDIOlog 3-13
*.apr 2-11
*.asm B-30
*.aws 10-2, 10-9
*.brd 2-2
*.c B-28
*.cpp B-28
*.ini 10-2, 10-8, 10-11
*.mk 11-9
*.prj 2-11
*.s B-50
*.sav 6-38, 12-18
*.src B-50
Fill Memory with Pattern
dialog 7-18
Find
dialog 13-15
Find in Files
c++ class definition 8-16
dialog 8-16, 13-7
text searching 13-7
Find Menu
in File Editor 12-15
Flash
definition files 7-6
FME files 7-7
setting interactively 7-6
Flash Memory Control
dialog 7-9, 7-10
Flash METHOD files
see Flash
Force scope 3-4
Formatting text 12-11
Frame pointer 6-24
Function List
dialog 8-6

G
Genmake location file 11-19
Global options
in workspace 10-2, 10-11
Glossary Glossary-1
Go button
in the Code window 3-6
Go until Return button
in the Code window 3-7

H
Hardware breakpoints
see Breakpoints
Heap 5-12
see Memory
High-level Step Into button
in the Code window 3-6
High-level Step Over button
in the Code window 3-7
Home
see Directories
HW Break if in Range
dialog 4-30
HW Break if X, then if Y
dialog 4-31
HW Break on Data Value match
dialog 4-32
HW While in func/range, Break if X
Images
auto-set PC on load 2-4
building 11-43
connecting on loading 2-6
debugging multiple images 2-17
disconnecting whilst loaded 2-20
load settings in project 2-13
loading 2-2, 5-5
loading from a user-defined project 2-2
loading from the command line 2-5
loading from the Load File to Target dialog box 2-3
loading from the Process Control pane 2-5
loading multiple 2-17
loading progress 2-7
managing 2-8
passing arguments on loading 2-4, 2-7
passing target name on loading 2-4
Processor State 2-7
quick loading 2-5
reloading 2-19, 2-20
removing all details 2-20
runtime indicator 2-7
set PC to entry point on load 2-4
setting PC manually 2-18
setting PC on load 2-4
symbol data 2-16
unloading 2-19
unloading and auto-projects 2-20
unloading and projects 11-28
viewing in the Code window 2-8
viewing in the Process Control pane 2-9
Include files 3-13
from the Debug menu 3-15
in projects 11-65
Journal files 3-13
Log files 3-13
STDIOLog files 3-13
Insert Template
dialog 12-21
Intended audience viii
Interactive Memory Setting
dialog 7-12
Interactive Register Setting
dialog 7-14
Interworking
see Projects
JOULOG_UNBUF 3-13
Journal files
at start-up 3-14, 3-16
closing 3-15
in the Status line 3-14
output buffering 3-13
viewing 3-15
Jump to Function
dialog 13-10
Library files
building 11-43
in projects 11-64
Library project 11-4
creating 11-31
properties groups 11-49
see also Projects
Line numbering
reset original 12-16
resetting in build 11-42
showing 12-16
using original 11-42, 12-16
Linker command files
generating 5-13
Linux
changes to breakpoints C-19
changes to debug views C-18
changes to the Code window C-22
changes to the desktop C-22
changes to the Pane controls C-23
Code window C-22
column resizing C-20
configuration files for RealView ARMulator ISS C-13
configuration files for Simulator Broker C-13
connecting to a remote host emulator C-28
connecting to a remote host simulator C-26
connecting using the CLI C-18
Connection Control window C-11
desktop links C-7
disconnecting remote connections C-31
evironment variables C-4, C-6
following examples C-2, C-23
Front Panel C-16
global configuration options C-19
Gnome C-6
home directory C-5
installing RealView Debugger C-3, C-6
KDE C-6
named connections C-18
de link controls C-23
RealView Network Broker C-24, C-25
remote connections C-24
remote hosts C-25
RVBROKER C-25
source coloring C-20
target configuration C-10
using a hosts file C-31
using DSP C-15
using RealView ARMulator ISS C-10
using Trace C-14
XML documentation C-7
Load File to Target
dialog 2-3
Loading
appending 2-4
images 2-2, 5-5
multiple images 2-17
replace image 2-4
Log files
at start-up 3-14, 3-16
closing 3-15
in the Status line 3-14
output buffering 3-13
viewing 3-15
Low-level Step Into button
Index

in the Code window 3-7
Low-level Step Over button
in the Code window 3-7

M

Macros 9-2
attaching 9-6
body 9-19
calling 9-4, 9-12
calling from the Debug menu 9-12
comments 9-20
conditional statements 9-20
copying 9-11
creating 9-7
defining 9-4
declaration 9-18
deleting 9-11
ing 9-9
in symbol table 9-12
including a definition file 9-4, 9-12
language 9-18
local symbols 9-20
precedence 9-5
predefined 9-26
prompting user 9-17
properties 9-3
return values 9-5
source patching example 9-13
stopping 9-6
terminator 9-19
using debugger commands 9-3
using with breakpoints 4-21, 9-5
viewing 9-8
MAKEFILE group B-80

Makefiles
contents 11-23
Custom projects 11-9, 11-32, B-82
generated 11-22
generation templates 11-9
Library projects 11-8
see also Projects
Standard projects 11-8
user-defined 11-9, 11-32, B-82

Memory
changing contents 6-17, 6-20
data formats 6-16
disabling memory mapping 5-4
display colors 6-17
downloading to a file 7-15
editing map entries 5-5
enabling memory mapping 5-4
filling with a pattern 7-18
formatting options 6-13
generating linker command files 5-13
interactive operations 7-2
interactive operations examples 7-12
interactive operations from the Debug menu 7-3
interactive operations from the Memory pane 7-5
interactive operations from the Stack pane 7-5
map configuration 5-8
mapping 5-2
setting interactively 7-12
setting stack and heap 5-12
setting top of memory 5-12
specifying range 7-16
verifying against a file 7-17
viewing contents 6-12
viewing for multiple targets 6-21
viewing the map 5-7
working with Flash 7-6
Memory map
display colors 5-8
editing 5-5
setting up 5-5
update based on processor registers 5-11

Menus
Break/Tracepoints 4-36
Complex Breakpoints 4-30
C++ Class 8-16
C++ Function 8-16
Debug 3-9
Editing Controls 12-4
Execution Control 3-10, 3-11
File 2-3
Find 8-2, 13-15
Memory/Register Operations 7-3
New Actions 4-19
New Qualifiers 4-18
Pane 4-37
Project 11-12
Settings options 10-4
Simple Breakpoints 4-10
Source control 14-3
Tools 10-8, 11-15
View 2-5
Workspace 10-3

Minimize button
in Editor Buffer List 12-7

Mode
ex in vi mode 12-26
vi in the File Editor 12-26
Module/File List
dialog 8-3

MS-DOS
applications in projects 11-59
names in projects 11-58

Multi-ICE
channel viewers 6-2
DCC 6-2
DCC semihosting 6-2, 6-11
Standard semihosting 6-2

N

Named breakpoint
see Breakpoints

Network Broker 1-6

New/Edit Favorite
dialog 4-41

Object files
building 11-43
in projects 11-62

Object

closing open 10-5
Open project list 11-81
Options window 10-8

P

Pair Matching
dialog 13-12
see Searching

Panes
<table>
<thead>
<tr>
<th>Break/Tracepoints</th>
<th>4-8, 4-33, 4-34</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Stack</td>
<td>6-28</td>
</tr>
<tr>
<td>File Editor</td>
<td>3-3</td>
</tr>
<tr>
<td>Memory</td>
<td>6-12</td>
</tr>
<tr>
<td>Output</td>
<td>13-8</td>
</tr>
<tr>
<td>Process Control</td>
<td>2-9</td>
</tr>
<tr>
<td>Register</td>
<td>6-3</td>
</tr>
<tr>
<td>Stack</td>
<td>6-24</td>
</tr>
<tr>
<td>Watch</td>
<td>6-32</td>
</tr>
<tr>
<td>PC</td>
<td>2-4</td>
</tr>
<tr>
<td>see also</td>
<td>Program Counter</td>
</tr>
<tr>
<td>Persistence info</td>
<td></td>
</tr>
<tr>
<td>favorites</td>
<td>4-41, 6-38</td>
</tr>
<tr>
<td>in Editing Controls</td>
<td>12-16</td>
</tr>
<tr>
<td>recent projects</td>
<td>11-13</td>
</tr>
<tr>
<td>Postlink commands</td>
<td></td>
</tr>
<tr>
<td>in projects</td>
<td>11-65</td>
</tr>
<tr>
<td>Prelink commands</td>
<td></td>
</tr>
<tr>
<td>in projects</td>
<td>11-65</td>
</tr>
<tr>
<td>Process</td>
<td></td>
</tr>
<tr>
<td>in Process Controlpane</td>
<td>2-10, 2-18</td>
</tr>
<tr>
<td></td>
<td>multiple images 2-17</td>
</tr>
<tr>
<td>Process Control</td>
<td></td>
</tr>
<tr>
<td>properties</td>
<td>2-11</td>
</tr>
<tr>
<td>tabs</td>
<td>2-9</td>
</tr>
<tr>
<td>Process Control pane</td>
<td></td>
</tr>
<tr>
<td>viewing projects</td>
<td>11-84</td>
</tr>
<tr>
<td>Processor</td>
<td></td>
</tr>
<tr>
<td>reset</td>
<td>2-19</td>
</tr>
<tr>
<td>resetting</td>
<td>2-19</td>
</tr>
<tr>
<td>Processor Events</td>
<td>see Breakpoints</td>
</tr>
<tr>
<td>Program Counter</td>
<td>2-4, 3-3</td>
</tr>
<tr>
<td>locating</td>
<td>3-4</td>
</tr>
<tr>
<td>red box</td>
<td>3-3</td>
</tr>
<tr>
<td>reset to entry point</td>
<td>3-4</td>
</tr>
<tr>
<td>see also</td>
<td>Scope</td>
</tr>
<tr>
<td>setting on load</td>
<td>2-4</td>
</tr>
<tr>
<td>Project Control</td>
<td>dialog 1-81</td>
</tr>
<tr>
<td>PROJECT group</td>
<td>B-3, B-8</td>
</tr>
<tr>
<td>Command_Open_Close</td>
<td>B-6</td>
</tr>
<tr>
<td>Modification_History</td>
<td>B-7</td>
</tr>
<tr>
<td>Project Properties</td>
<td></td>
</tr>
<tr>
<td>ASSEMBLE group</td>
<td>B-46</td>
</tr>
<tr>
<td>auto-project groups</td>
<td>11-49</td>
</tr>
<tr>
<td>BUILD group</td>
<td>B-60</td>
</tr>
<tr>
<td>BUILD_LIB group</td>
<td>B-72</td>
</tr>
<tr>
<td>COMPILE group</td>
<td>B-24</td>
</tr>
<tr>
<td>CONFIGURATION group</td>
<td>B-22</td>
</tr>
<tr>
<td>Container project groups</td>
<td>11-49</td>
</tr>
<tr>
<td>CUSTOM group</td>
<td>B-58</td>
</tr>
<tr>
<td>Custom project groups</td>
<td>11-49</td>
</tr>
<tr>
<td>default setting</td>
<td>11-46</td>
</tr>
<tr>
<td>Library project groups</td>
<td>11-49</td>
</tr>
<tr>
<td>List of Entries pane</td>
<td>11-47</td>
</tr>
<tr>
<td>MAKEFILE group</td>
<td>B-80</td>
</tr>
<tr>
<td>PROJECT group</td>
<td>B-3, B-8</td>
</tr>
<tr>
<td>SETTINGS group</td>
<td>B-9</td>
</tr>
<tr>
<td>Settings Values pane</td>
<td>11-47</td>
</tr>
<tr>
<td>Standard project groups</td>
<td>11-49</td>
</tr>
<tr>
<td>window</td>
<td>11-7, 11-46, 11-47</td>
</tr>
<tr>
<td>Projects</td>
<td></td>
</tr>
<tr>
<td>active</td>
<td>11-12</td>
</tr>
<tr>
<td>active build target configuration</td>
<td>11-75</td>
</tr>
<tr>
<td>active project</td>
<td>11-103</td>
</tr>
<tr>
<td>active project and binding</td>
<td>11-82, 11-97</td>
</tr>
<tr>
<td>active project on closing</td>
<td>11-110</td>
</tr>
<tr>
<td>adding files</td>
<td>11-14, 11-23, 11-24</td>
</tr>
<tr>
<td>adding library files</td>
<td>11-64</td>
</tr>
<tr>
<td>adding non-source files</td>
<td>11-62, 11-64</td>
</tr>
<tr>
<td>adding object files</td>
<td>11-62</td>
</tr>
<tr>
<td>adding paths for include files</td>
<td>11-65</td>
</tr>
<tr>
<td>adding prelink commands</td>
<td>11-65</td>
</tr>
<tr>
<td>adding prelink commands</td>
<td>11-65</td>
</tr>
<tr>
<td>adding source files</td>
<td>11-53</td>
</tr>
<tr>
<td>associated images</td>
<td>2-14, 11-4</td>
</tr>
<tr>
<td>autobinding</td>
<td>11-10, 11-91, 11-95, 11-96, 11-98</td>
</tr>
<tr>
<td>autobinding rules</td>
<td>11-92</td>
</tr>
<tr>
<td>auto-projects</td>
<td>2-11, 2-12, 11-5, 11-107</td>
</tr>
<tr>
<td>auto-projects in the Process Control pane</td>
<td>11-85</td>
</tr>
<tr>
<td>binding</td>
<td>11-10, 11-28, 11-90</td>
</tr>
<tr>
<td>binding ambiguity</td>
<td>11-95</td>
</tr>
<tr>
<td>binding in the Code window</td>
<td>11-95</td>
</tr>
<tr>
<td>binding types</td>
<td>11-90</td>
</tr>
<tr>
<td>bound</td>
<td>11-95</td>
</tr>
<tr>
<td>build errors</td>
<td>11-45</td>
</tr>
<tr>
<td>build model</td>
<td>11-41</td>
</tr>
<tr>
<td>build rules</td>
<td>11-41</td>
</tr>
<tr>
<td>build target configurations</td>
<td>11-7, 11-72</td>
</tr>
<tr>
<td>build tools</td>
<td>11-13, 11-18, 11-57</td>
</tr>
<tr>
<td>build tools for user-defined projects</td>
<td>11-7</td>
</tr>
<tr>
<td>building</td>
<td>11-18</td>
</tr>
<tr>
<td>building an application</td>
<td>11-41</td>
</tr>
<tr>
<td>building images</td>
<td>11-43</td>
</tr>
<tr>
<td>building library files</td>
<td>11-43</td>
</tr>
<tr>
<td>building object files</td>
<td>11-43</td>
</tr>
<tr>
<td>building using the Build dialog box</td>
<td>11-44</td>
</tr>
<tr>
<td>changing active build target configuration</td>
<td>11-75</td>
</tr>
<tr>
<td>changing build tools</td>
<td>11-57</td>
</tr>
<tr>
<td>changing location of object files</td>
<td>11-56</td>
</tr>
<tr>
<td>changing properties</td>
<td>11-52</td>
</tr>
<tr>
<td>closing</td>
<td>2-15, 11-13, 11-27, 11-109</td>
</tr>
<tr>
<td>closing and defining the active project</td>
<td>11-110</td>
</tr>
<tr>
<td>collecting source files</td>
<td>11-88</td>
</tr>
<tr>
<td>compiler options in interworking</td>
<td>11-69</td>
</tr>
<tr>
<td>compiling source files</td>
<td>11-55</td>
</tr>
<tr>
<td>Container</td>
<td>11-34</td>
</tr>
<tr>
<td>Container projects</td>
<td>11-105</td>
</tr>
<tr>
<td>copy of existing</td>
<td>11-35</td>
</tr>
<tr>
<td>creating</td>
<td>11-29, 11-35</td>
</tr>
<tr>
<td>Custom</td>
<td>11-32</td>
</tr>
<tr>
<td>customizing the build</td>
<td>11-59</td>
</tr>
<tr>
<td>default active project</td>
<td>11-82, 11-104</td>
</tr>
<tr>
<td>default binding</td>
<td>11-10, 11-21, 11-90, 11-100</td>
</tr>
<tr>
<td>default binding rules</td>
<td>11-91</td>
</tr>
<tr>
<td>defining build tools</td>
<td>11-19</td>
</tr>
<tr>
<td>defining in connection properties</td>
<td>11-11</td>
</tr>
<tr>
<td>defining in workspace</td>
<td>10-6, 11-11</td>
</tr>
<tr>
<td>deleting</td>
<td>11-6</td>
</tr>
<tr>
<td>disabling a group</td>
<td>11-50</td>
</tr>
<tr>
<td>displaying code sizes in interworking</td>
<td>11-71</td>
</tr>
<tr>
<td>effects of binding</td>
<td>11-99</td>
</tr>
<tr>
<td>effects of connecting</td>
<td>11-100, 11-110</td>
</tr>
<tr>
<td>effects of disconnecting</td>
<td>11-100, 11-110</td>
</tr>
<tr>
<td>effects of unbinding</td>
<td>11-100</td>
</tr>
<tr>
<td>enabling a group</td>
<td>11-50</td>
</tr>
<tr>
<td>excluding source files from build</td>
<td>11-55</td>
</tr>
</tbody>
</table>
forcing binding   11-97
forcing unbinding   11-99
generated makefiles   11-22
genmake.loc   11-19
group properties   11-49
in Process Control pane   11-84
interworking ARM and Thumb   11-68
Library   11-31
linker options   11-67
linking object files   11-65
loading multiple images   11-107
makefiles   11-8
makefiles layout templates   11-9
MAXLINELENGTH in makefile   11-6
merging auto-project settings   11-29, 11-36
mixing auto-projects and user-defined projects   11-107
multiple device names in autobinding   11-95
multiple devices in autobinding   11-94
multiple projects   11-82, 11-102
nesting container projects   11-34, 11-107, B-8
no build   11-32
open project list   11-81
opening   11-13, 11-21
organizing   11-6
outputting a build message   11-59
partial matching in autobinding   11-93
project base directory   11-6
Project Properties window   11-7
properties   11-23, 11-46
see also Appendix B
rebuilding   11-14
re-including source files into a build   11-56
removing source files   11-54
scatter loading   11-67
self-contained   11-6
size   11-6
specific device binding   11-91, 11-93
specification   11-3
specifying compilers   11-70
specifying subprojects list   11-105
Standard   11-31
Tools menu   11-15
types   11-4
unbound   11-95
updating dependencies   11-14
upgrading to a new toolchain   11-22, 11-25, 11-27
user-defined   11-3
user-defined projects in the Process Control pane   11-85
using breakpoints   11-66
using MS-DOS applications   11-59
using MS-DOS names   11-58
using the Configuration Summary window   11-50
using the Project Control dialog box   11-82
using your own makefile   11-32
viewing debugger internals   6-9
viewing for multiple targets   6-7
viewing properties   11-46
viewing contents   6-3
viewing debugger internals   6-9
viewing for multiple targets   6-7
viewing properties   11-46
working with   11-2
PVCS   14-2

R

RealView ARMulator ISS
see Linux
see Solaris
RealView Debugger
arguments   1-2
connecting to a target   2-2
debugger internals   6-9
directories   1-8
environment variables   1-4
image loading   5-5
memory mapping   5-2
predefined macros   9-26
starting   1-2
target configuration   2-2
using Linux   C-2
using macros   9-2
using Solaris   C-2
RealView Network Broker
arguments   1-6
using Linux   C-24
using Solaris   C-24
Redo command
in File Editor   12-13
Registers
changing contents   6-6
defining custom registers   6-11
display colors   6-5
formatting options   6-5
interactive operations   7-2
interactive operations examples   7-12
interactive operations from the Debug menu   7-3
setting interactively   7-14
upgrading to a new toolchain   11-22, 11-25, 11-27
user-defined   11-3
user-defined projects in the Process Control pane   11-85
using breakpoints   11-66
using MS-DOS applications   11-59
using MS-DOS names   11-58
using the Configuration Summary window   11-50
using the Project Control dialog box   11-82
using your own makefile   11-32
viewing in interworking   11-68
viewing properties   11-46
viewing contents   6-3
viewing debugger internals   6-9
viewing for multiple targets   6-7

Regular expressions
replacing in text   13-17
searching in text   13-4, 13-17, 13-18
Reloading
images   2-20
Re-Open command
in File Editor   12-13
Replacing
global changes   13-5
regular expressions   13-17
text   13-5
text in File Editor   13-1, 13-5
text in File editor   13-4
ReScan button
in Editor Buffer List   12-7
Restore button
in Editor Buffer List   12-7
Run-time stack
see Stack
RVDEBUG   3-16
RVDEBUG_HOME   1-5
RVDEBUG_INSTALL   1-4
RVDEBUG_SHARE   1-5

S

Save button
in Editor Buffer List   12-7
Save Multiple
dialog   12-9
Scatter loading
see Projects
Index

Scope  3-3
autoscope  3-3
blue pointer  3-4
context  3-3
forcing  3-4
in browsers  8-2
PC  3-3
red box  3-3, 3-4, 3-6
scoping to a file  8-4
scoping to a function  8-8
scoping to a module  8-4
Scope controls  6-26, 6-31
Searching
direction  13-4
enclosing pair  13-14
Find in Files  13-7
Find in menu  13-15
function definitions in text  13-10, 13-11
functions in text  13-10
nested pair matching example  13-14
pair matching  13-12
regular expressions  13-4, 13-17
regular expressions with special operators  13-17
setting options  13-15
source files  3-16, 12-17
text in File Editor  13-1, 13-3
text in multiple tabs  13-4
viewing results  13-8
viewing results in FileFind tab  13-9
Select File to Include Commands from dialog  3-15
Select File to Journal to dialog  3-14
Select File to log the STDIO to dialog  3-14
Select File to Log to dialog  3-14
Select Project to Copy from dialog  11-35
Select Project to Open dialog  11-21
Semihosting  6-2
DCC  6-2
Standard  6-2
Set Address/Data Break/Tracepoint dialog  4-10
Settings
save on exit  10-4
SETTINGs group  B-9
Auto_Set_Breaks  B-14
Image_load  B-12
merging for auto-projects  11-37
Named_Breaks  B-16
Runtime_Control  B-18
Vectors  B-20
Settings options  10-4
Settings: List Manager
dialog  11-63, 11-64
Simple Break if X
dialog  4-13
Simple Break if X, N times
dialog  4-14
Simple Break if X, when Y is True
dialog  4-14
Simple breakpoints
see Breakpoints
Software breakpoints
see Breakpoints
Solaris
changes to breakpoints  C-19
changes to debug views  C-18
changes to the Code window  C-22
changes to the desktop  C-22
changes to the Pane controls  C-23
Code window  C-22
column resizing  C-20
configuration files for RealView ARMulator ISS  C-13
configuration files for Simulator Broker  C-13
connecting to a remote host emulator  C-28
connecting to a remote host simulator  C-26
connecting using the CLI  C-18
Connection Control window  C-11
disconnecting remote connections  C-31
environment variables  C-4, C-5
following examples  C-2, C-23
Front Panel  C-16
global configuration options  C-19
home directory  C-5
installing RealView Debugger  C-3, C-5
named connections  C-18
pane controls  C-23
RealView Network Broker  C-24, C-25
remote connections  C-24
remote hosts  C-25
RVBROKER  C-25
source coloring  C-20
target configuration  C-10
using a hosts file  C-31
using DSP  C-15
using RealView ARMulator ISS  C-10
using Trace  C-14
Source control
see also Version Control
Source control button
in the Code window  14-3
Source control settings
custom commands  14-11
in the workspace  14-9
Source files
collecting  11-88
excluding from build  11-55
in projects  11-53
re-including in a build  11-56
Source Search Paths
dialog  3-16
Specific device binding
see Projects
Stack
breakpoints  6-26
changing contents  6-26
context controls  6-26, 6-31
formatting options  6-25
FP  6-24
in the Stack pane  6-23, 6-24
pointer  6-23
see also Memory
SP  6-24
Stack down button
in the Code window  6-26
Stack pointer  6-24
Stack up button
in the Code window  6-26
Standard project  11-4
creating  11-31
properties groups  11-49
see also Projects
Index

Starting
RealView Debugger  1-2
STDIOlog files
at start-up  3-14, 3-16
closing  3-15
in the Status line  3-14
output buffering  3-13
viewing  3-15
Stop Execution button
in the Code window  3-8
Structure of this book  ix
Symbols
loading  2-4, 2-16

T
Target
see Debug target
Target configurations
see Build target configurations
Templates
creating  12-22
editing  12-22
examples  12-23
inserting  12-21
Terminology  Glossary-1
Text
adding  12-10
changing case  12-11, 12-12
coloring  12-12
configuring searches  13-15
creating  12-10
deleting  12-11
direction when searching  13-4
dragging and dropping  12-13
editing  12-9
enclosing pair  13-14
formatting  12-11
jumping back from functions  13-11
jumping to functions  13-10
moving between windows  12-14
nested pair matching example  13-14
pair matching  13-12
replace regular expressions  13-17
replacing  13-4, 13-5
search regular expressions  13-4, 13-17
searching  13-3, 13-4
searching across multiple files  13-7
searching and replacing  13-1
searching for functions  13-10
searching for selected text  13-6
searching multiple tabs  13-4
searching options  13-15
searching regular expressions  13-18
searching using the Find menu  13-15
sensitivity in replacing  13-5
sensitivity in searches  13-5
shift left  12-12
shift right  12-12
special operators in regular expressions  13-17
Undo command in replacing  13-5
undoing changes in File Editor  12-13
viewing search results  13-8
viewing search results in FileFind tab  13-9
Toolbars
Actions  11-17
customizing  10-11
Editing  14-3
Toolchain
genmake.loc  11-19
Tracepoints  3-10
in the Break/Tracepoints pane  4-33, 4-34
in the Set Address/Data Break/Tracepoints dialog  4-10
markers  4-7
type  4-12

U
Undo command
in File Editor  12-13
in text replacing  13-5
Unloading
images  2-19
Upgrade Project ToolChain
dialog  11-25
Upload/Download file from/to Memory dialog  7-15
User-defined projects
overview  11-3
see also Projects
User’s comments  xv

V
Variable List
dialog  8-10
Veneers  11-68
see also Projects
Version Control  14-2
changing the tool used  14-2
ClearCase  14-2
defining the tool  14-2
in Code window  14-3
in File Editor  12-3
in workspace  14-8
no tool identified  14-2
PVCS  14-2
setting a prompt  14-7
Source control button  14-2
Source control menu  14-3
SrcCtrl tab  14-7
under UNIX  14-2
WinCVS  14-3
WinCVS under Windows  14-3
Vi
in the File Editor  12-26
VISUAL  12-20

W
Watches
creating a favorite  6-38
editing  6-37
in the Watch pane  6-32
managing  6-35
saving as favorites  6-39
setting  6-32
setting from favorites  6-37
viewing expressions  6-34
Who should read this book  viii
WinCVS  14-3
Window
Configuration Summary  11-50
Windows
Build-Tool Properties  11-13, 11-19
Index

Editor 12-19
Options 10-8
Project Properties 11-13
Workspace Options 10-8

Workspace
ALL 10-11
see also Appendix A
Asm_type setting A-7
Backup settings A-12
Board_file setting A-4
Button settings A-7
changing settings 10-13
closing 10-5
closing objects 10-5
CODE 10-11
see also Appendix A
Command settings A-2
connection details 10-12
copying entries 10-14
creating an empty file 10-7
current settings 12-17
custom source control commands 14-10, 14-11
cutting entries 10-17
DEBUGGER 10-11
see also Appendix A
debugger settings 10-14
defining Code windows 10-13
deleting settings 10-17
Disassembler settings A-3
Edit settings 12-17, A-13
global options 10-2, 10-11
in Tools menu 10-8
in Workspace Options window 10-8
initializing 10-2
List of Entries pane 10-10
not using 10-2
open projects 10-6
opening 10-3, 10-4
pasting entries 10-15
Pos_size settings A-6
project load list 10-6
PROJECTS 10-11
see also Appendix A
resetting entries 10-17
same on start-up 10-4
saving 10-3
Search settings A-11
settings 10-2, 10-11

see also Appendix A
settings conflict 10-5, 10-11
settings file 10-11
settings saved 10-2
Settings Values pane 10-10
source control settings 14-8, 14-9
Source-coloring settings A-9
Src_ctrl settings A-13
start-up options 10-3
Tab_conv settings A-12
Text settings A-10
undoing changes 10-17
using 10-2
viewing settings 10-8
window internals 10-11
Workspace menu
in the Code window 10-3
Workspace Options window 10-9
Index