RealView® Debugger

Version 1.8

User Guide

ARM®

ARM DUI 0153G

RealView Debugger
User Guide

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Change
April 2002	A	RealView Debugger v1.5 release
September 2002	B	RealView Debugger v1.6 release
February 2003	C	RealView Debugger v1.6.1 release
September 2003	D	RealView Debugger v1.6.1 release for RVDS v2.0
January 2004	E	RealView Debugger v1.7 release for RVDS v2.1
December 2004	F	RealView Debugger v1.8 release for RVDS v2.2
May 2005	G	RealView Debugger v1.8 SP1 release for RVDS v2.2 SP1

Proprietary Notice

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

Contents
RealView Debugger User Guide

Preface

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

About this book

This book describes how to use RealView Debugger to debug applications and images:

- a detailed description of how to use RealView Debugger to debug images using a range of debug targets, including examples
- a description of the built-in features of RealView Debugger, such as workspaces and macros
- appendixes containing reference information for the software developer
- a glossary of terms for users new to RealView Debugger.

Intended audience

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

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

Before you start

It is recommended that you read *RealView Debugger v1.8 Essentials Guide* before starting to use this book. In particular, read the chapter that describes 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 sample programs and projects stored in the `\Examples` directory in your RealView Developer Suite root installation.

In general, examples use *RealView ARMulator® ISS* (RVISS) to simulate an ARM architecture-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 of how to start using RealView Debugger on your workstation.

Chapter 2 ***Working with Images***

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

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

Chapter 5 ***Memory Mapping***

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

Chapter 6 ***Working with Debug Views***

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

Chapter 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 greater control over your application software.

Chapter 8 ***Working with Browsers and Favorites***

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 create 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.

Appendixes and Glossary

Appendix A *Workspace Settings Reference*

Read this appendix for details of how to set options to configure your working environment using RealView Debugger workspaces. This appendix must be read in association with Chapter 10 *Configuring Workspace Settings*.

Appendix B *RealView Debugger on Sun Solaris and Red Hat Linux*

Read this appendix for details of how to use RealView Debugger on Sun Solaris and Red Hat Linux. This appendix contains corrections and additions to the documentation suite.

Glossary

An alphabetically arranged glossary defines the special 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.
`mono`space	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 v1.8 Essentials Guide* (ARM DUI 0181)
- *RealView Debugger v1.8 Target Configuration Guide* (ARM DUI 0182)
- *RealView Debugger v1.8 Command Line Reference Guide* (ARM DUI 0175)
- *RealView Debugger v1.8 Extensions User Guide* (ARM DUI 0174)
- *RealView Debugger v1.8 Project Management User Guide* (ARM DUI 0227).

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

For details on using RVISS, see the following documentation:

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

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

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

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

- *RealView ICE User Guide* (ARM DUI 0155)
- *Multi-ICE® User Guide* (ARM DUI 0048)
- *ARM MultiTrace™ User Guide* (ARM DUI 0150).

Other publications

For a comprehensive introduction to ARM architecture see:

Steve Furber, *ARM system-on-chip architecture* (2nd edition, 2000). Addison Wesley, ISBN 0-201-67519-6.

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

Jeffrey E. F. Friedl, *Mastering Regular Expressions, Powerful Techniques for Perl and Other Tools*, 1997. O'Reilly & Associates, Inc. ISBN 1-56592-257-3.

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

Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language* (2nd edition, 1989). Prentice-Hall, ISBN 0-13-110362-8.

For more information about the JTAG standard, see:

IEEE Standard Test Access Port and Boundary Scan Architecture (IEEE Std. 1149.1), available from the IEEE (`http://www.ieee.org`).

For more information about CEVA-Oak, CEVA-TeakLite, and CEVA-Teak processors from CEVA, Inc. see `http://www.ceva-dsp.com`.

For more information about the ZSP400 and ZSP500 processors from the ZSP division of LSI Logic see `http://www.zsp.com`.

Feedback

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

Feedback on RealView Debugger

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

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

Feedback on this book

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

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

General suggestions for additions and improvements are welcome.

Chapter 1
Starting to use RealView Debugger

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

Note

Where RealView Debugger panes are shown in this document, they are shown as floating panes. Floating panes show the name of the pane in the title bar. Docked panes do not have a title bar, and so do not show the title. See the *RealView Debugger v1.8 Essentials Guide* for more details on panes in RealView Debugger.

1.1 Starting RealView Debugger

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

- *Starting on Windows*
- *Starting on Sun Solaris and Red Hat Linux*
- *Starting from the command line*
- *Setting user-defined environment variables* on page 1-7.

1.1.1 Starting on Windows

To start RealView Debugger, select the following from the Windows **Start** menu:

Programs → ARM → RealView Developer Suite v2.2 → RealView Debugger v1.8.1

1.1.2 Starting on Sun Solaris and Red Hat Linux

To start RealView Debugger on Sun Solaris or Red Hat Linux, a RealView Debugger *Control Panel*, `RVDEBUGCP` is provided. See *Getting started with RealView Debugger* on page B-7 for more details on using `RVDEBUGCP`).

1.1.3 Starting from the command line

If your *RealView Developer Suite* (RVDS) environment is correctly configured, you can start RealView Debugger from the command line by typing `rvdebug` together with any arguments (see *Syntax of the rvdebug command* on page 1-3).

By default, the RealView Debugger starts in graphical user interface (GUI) mode. You must specify the `-cmd` option to start RealView Debugger in command line interface (CLI) mode.

Note

If you are having problems starting RealView Debugger, see the *RealView Developer Suite Getting Started Guide* for details of how to fix problems with your RVDS environment.

Syntax of the rvdebug command

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

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

where:

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

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

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

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

Can be replaced with `-b`.

Note

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

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

`-cmd` Runs the RealView Debugger in headless debugger mode, where you can use CLI commands to carry out debugging tasks. This enables you to interact with the debugger without using the RealView Debugger GUI. See the *RealView Debugger v1.8 Essentials Guide* for details on getting started with the headless debugger. Also, see the *RealView Debugger v1.8 Command Line Reference Guide* for details on the CLI commands.

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

Note

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

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

Note

This option is only available under Windows. On Sun Solaris and Red Hat Linux platforms, RealView Debugger always uses the home directory as specified by the `$RVDEBUG_HOME` environment variable.

See *Defining the home directory on Windows* on page 1-11 for details.

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

Use `-inc`:

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

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

`@endpoint_connection@access_provider`

For example, `@new_arm@localhost` for RVISS.

`-install` Specifies the installation directory where this differs from the default installation. This must point to the following directory:

`install_directory``\RVD\Core\...\`*`platform`*

On Windows systems, this is then used to define the location of the default RealView Debugger home directory when the debugger runs for the first time.

This overrides the environment variable `RVDEBUG_INSTALL`, and must be used if `RVDEBUG_INSTALL` is not set.

`-jou` Runs a RealView Debugger session with the specified journal file open for writing. You can optionally include `=`, for example `-jou=`*`filename`*. Can be replaced with `-j`.

`-log` Runs a RealView Debugger session with the specified log file open for writing. You can optionally include =, for example `-log=filename`. Can be replaced with `-l`.

`-nologo` Runs a RealView Debugger session without displaying a splash screen.

`-stdiolog` Runs a RealView Debugger session with the specified STDIOlog file open for writing. You can optionally include =, for example `-stdiolog=filename`. Can be replaced with `-s`.

`-user` Specifies the user ID in the RealView Debugger home directory used for the debugging session. Where this is not specified, the default Windows login is used.

See *Defining the home directory on Windows* on page 1-11 for details.

Examples

The following examples show how to use some of the command line options when starting RealView Debugger from the command line on Windows. The examples assume that RealView Debugger is installed on the C drive, and use a working directory on the D drive.

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

  ```
  C:\ARM\RVD\Core\...\win_32-pentium\bin\rvdebug.exe
  -home="D:\rvd_work\home\my_user_name”
  ```

- To start RealView Debugger and specify a workspace:

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

- To start RealView Debugger without loading a workspace:

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

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

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

- To start RealView Debugger with a connection to RVISS, and an image loaded that takes two arguments:

  ```
  C:\ARM\RVD\Core\...\win_32-pentium\bin\rvdebug.exe
  -init=@new_arm@localhost -exec “D:\rvd_work\images\my_image.axf;;100 200”
  ```

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.

1.1.4 Setting user-defined environment variables

You can set user-defined environment variables to reconfigure the RealView Debugger environment to your particular requirements.

Overriding the default home directory

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

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

Specifying a shared location

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

Target 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. This section includes:

- *RealView Connection Broker operating modes*
- *Starting RealView Connection Broker on Windows*
- *Starting RealView Connection Broker from the command line* on page 1-9
- *Using RealView Network Broker* on page 1-10.

1.2.1 RealView Connection Broker operating modes

RealView Connection Broker operates in two modes:

Local Operating as RealView Simulator 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` vehicle, ready to connect to a simulator, RealView Debugger starts RealView Connection Broker in local mode to manage your connection.

See the chapter that describes configuring custom connections in *RealView Debugger v1.8 Target Configuration Guide* for examples of how to set up your own connections.

1.2.2 Starting RealView Connection Broker on Windows

To start RealView Connection Broker, select the following from the Windows **Start** menu:

Programs → ARM → RealView Developer Suite v2.2 → RealView Network Broker v1.8.1

This starts RealView Connection Broker in remote mode.

1.2.3 Starting RealView Connection Broker from the command line

If your *RealView Developer Suite* (RVDS) environment is correctly configured, you can start RealView Connection Broker from the command line by typing `rvbroker` together with any arguments (see *RealView Connection Broker command line syntax*).

By default, RealView Connection Broker starts as RealView Simulator Broker. You must specify the `remote` option to start RealView Network Broker.

Note

If you are having problems starting RealView Connection Broker, see the *RealView Developer Suite Getting Started Guide* for details of how to fix problems with your RVDS environment.

RealView Connection Broker command line syntax

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

```
rvbroker -install=pathname [port_num] [remote]
```

where:

`-install` Specifies the RealView Debugger installation directory where this differs from the default installation.

This overrides the environment variable `RVDEBUG_INSTALL`, and must be used if `RVDEBUG_INSTALL` is not set.

`port_num` Specifies the TCP/IP port number to use. `0` is the default port number. You can use a port number greater than zero for remote connections.

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

- if specified, RealView Connection Broker runs as RealView Network Broker
- if not specified, RealView Connection Broker runs as RealView Simulator Broker.

1.2.4 Using 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 Sun Solaris or Red Hat Linux 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. Start RealView Network Broker as described in:
   - *Starting RealView Connection Broker on Windows* on page 1-8
   - *Starting RealView Connection Broker from the command line* on page 1-9.

Note

If you end a debugging session, and close down RealView Debugger, this does not terminate RealView Network Broker on the remote workstation. You must shut this 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 that describes working with remote targets in *RealView Debugger v1.8 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 on Windows*
- *The home directory on Sun Solaris or Red Hat Linux* on page 1-12
- *Using the examples directories* on page 1-12.

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 sequence of tests to define the installation directory:

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

1.3.2 Defining the home directory on Windows

RealView Debugger requires a home directory to store user-specific settings and configuration files. This is not the same as your Windows home directory.

On Windows, the location of this directory depends on the environment variables set, and the command line arguments used, when RealView Debugger starts. It uses the following sequence of tests to define the home directory:

1. The `-home` command line argument (only available under Windows), if used (see *Starting from the command line* on page 1-2).
2. The `RVDEBUG_HOME` environment variable, if set (see *Setting user-defined environment variables* on page 1-7).
3. The `-user` command line argument, if used (see *Starting from the command line* on page 1-2). This is then used to specify the user ID in the home directory, for example set `USER=my_user_name` to specify the home directory:

   `install_directory``\RVD\Core\...\home\my_user_name`.
4. Your default Windows login, for example:

   `install_directory``\RVD\Core\...\home\`*`WinLogID`*.

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

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

install_directory\RVD\Core\...\home*user_name*

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

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

1.3.3 The home directory on Sun Solaris or Red Hat Linux

On Sun Solaris or Red Hat Linux, your home directory is in ~/rvd.

1.3.4 Using the examples directories

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

install_directory\RVDS\Examples\...*project*

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

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

Chapter 2
Working with Images

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

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 begin debugging images you must connect to a suitably configured debug target.

RealView Debugger uses a *board file* to access information about the debugging environment and the debug targets available to you, for example how memory is mapped. See *RealView Debugger v1.8 Target Configuration Guide* for details of how to customize your targets using your board file.

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

Select **Target → Connect to Target...** from the main menu to display the Connection Control window to make your first connection. For details on using this window, see the chapter that describes getting started with RealView Debugger in *RealView Debugger v1.8 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* on page 2-3
- *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 2-6
- *Loading and runtime visualization* on page 2-8.

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

Note

When you are using RealView ARMulator® ISS software simulator to simulate an ARM® architecture-based debug target:

- You must load an image, or write to the PC, to begin execution.
- Loading an image 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.

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.

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

install_directory\RVDS\Examples\...\dhrystone\DebugRel\dhrystone.axf

Note

If you load an image, built as part of a user-defined project, without opening the project, this does not give you access to all the project properties because these are unknown to RealView Debugger. In this case, RealView Debugger creates an in-memory project, or uses the saved *auto-project* file (see *Working with auto-projects* on page 2-13 for details).

2.1.2 Using the Load File to Target dialog box

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

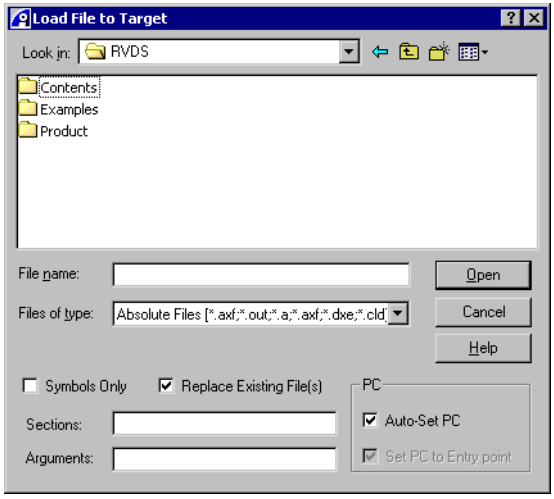

Figure 2-1 Load File to Target dialog box

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

Symbols Only

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

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

See *Working with symbols* on page 2-19 for more details.

Replace Existing File(s)

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

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

Sections:

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

A name entered here is then used as the argument to a `LOAD` command (see *Specifying the target connection and load instruction* on page 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 target connection and 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, if you select the check box **Replace Existing File(s)**.

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

Set PC to Entry point

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

This is the default if you select both:

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

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

Note

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

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 → Process Control** from the default Code window main menu to display the Process Control pane.

   With no image loaded, the pane only shows details about the debug target processor and the current location of the PC.

2. Right-click on the top line, the processor entry (for example `ARM7TDMI`), to display the **Process** context menu.

   With no image loaded, you can also display this menu from the `Image` entry.

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

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

2.1.4 Quick loading

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

Note

Ensure that the target device for the current connection, shown in the Code window title bar, matches the processor type of the image you are trying to load. If they do not match the load fails. For example, you cannot load an image built for an ARM architecture-based processor on a DSP.

In addition, make sure that the processor or architecture of the target matches the processor or architecture that you specified when you built the image. Although the image might load, it might not run correctly. For example, you cannot run an image built for an ARM966E-S processor on an ARM7TDMI target.

2.1.5 Loading from the command line

You can start RealView Debugger from the command line and specify an image to load automatically. You must also specify a target connection to use. The syntax for loading an image this way is:

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

where *pathname* specifies the image loaded and can also include image sections to be loaded and image arguments. See *Specifying the target connection and load instruction* on page 2-7 for more details.

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

```
rvdebug.exe -init=@new_arm@localhost -exec "C:\rvd_work\my images\my_image.axf"
```

This starts RealView Debugger, connects to RVISS, and issues a `load/pd/r` command to load the named image to your debug target. The `/pd` switch specifies that any error messages are to appear in a dialog box. The `/r` switch specifies that any image currently loaded on the chosen target is to be replaced by the specified image. For details of the `LOAD` command, see *RealView Debugger v1.8 Command Line Reference Guide*.

Note

For details on running RealView Debugger from the command line see Chapter 1 *Starting to use RealView Debugger*. Also see the chapter that describes getting started with the CLI in the *RealView Debugger v1.8 Essentials Guide*.

Connection prompt on load failure

Starting RealView Debugger in this way connects to the target, loads the specified image to that target, and updates the Code window. This is the same as *Using the Load File to Target dialog box* on page 2-3.

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

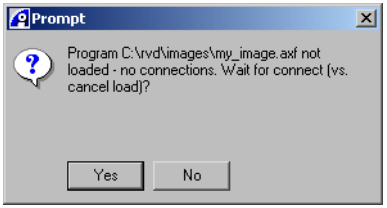

Figure 2-2 Connection prompt

Click:

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

No Starts the debugger but cancels the image loading operation.

Specifying the target connection and load instruction

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

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

where:

endpoint_connection
: Specifies the endpoint connection name.

access_provider
: Specifies the access provider name for the connection.

image.axf
: Specifies the image to be loaded.

sections
: Specifies the sections to load for the image, for example, ER_RO,ER_RW.

arg1 arg2 ...
: Specifies an optional, space-separated, list of arguments to the image.

Note

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

For details on specifying sections and arguments, see the LOAD command in *RealView Debugger v1.8 Command Line Reference Guide*.

Examples

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

- To establish a connection to the Simulator Broker interface of RVISS, and load an image with sections `ER_RO` and `ER_RW` and the argument `5000`:

  ```
  rvdebug.exe -init=@new_arm@localhost -exec
  "C:\rvd\images\my_image.axf;ER_RO,ER_RW;5000"
  ```

  If you want to supply arguments, but no sections, leave the sections blank, for example:

  ```
  "C:\rvd\images\my_image.axf;;5000"
  ```

- To establish a connection to the Simulator Broker interface of RVISS, and load an image without specific sections and without arguments:

  ```
  rvdebug.exe -init=@new_arm@localhost -exec C:\rvd\images\my_image.axf
  ```

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 Status line also shows the Processor status of the debug target:

- Where an image is loaded but not executing, the status shows `Stopped`.
- Where an image is running, the status shows `Running`, together with a moving progress.
- Where the current state of the target is unknown, the status shows `Unknown`. For example it might have been running when the connection was established or it might be disconnected.

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 2-11
- *Working with auto-projects* on page 2-13
- *Working with user-defined projects* on page 2-17.

The examples in this section assume that:

- you are using a Typical installation and that the software has been installed in the default location. If you have changed these defaults, or set the environment variable `RVDEBUG_INSTALL`, your installation differs from that described here.
- you are licensed to use multiprocessor debugging mode. However, where given, multiprocessor examples are only used for illustration. If you are licensed to work in single-processor mode your Code window differs from that shown here, for example some toolbar buttons are not available to you, but all the examples work as described.

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 on page 2-10.

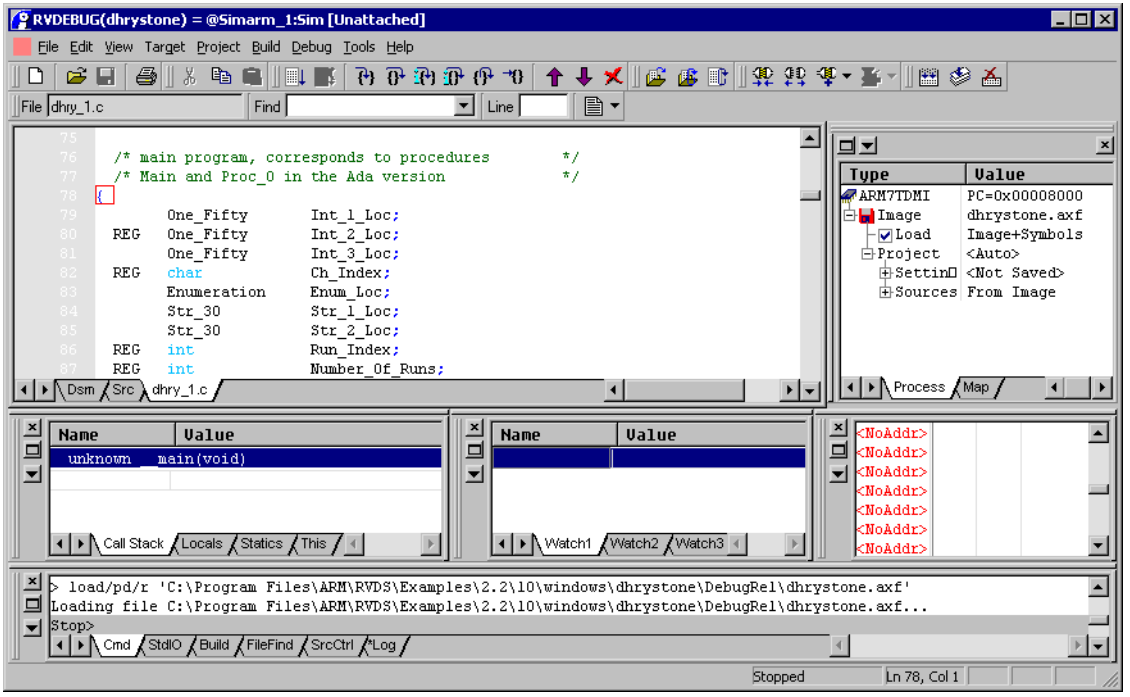

Figure 2-3 Code window with image loaded

Note

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

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

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

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

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

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

2.2.2 Viewing image details in the Process Control pane

By default, the side pane to the right of the File Editor pane is the Process Control pane. However, if you no longer have the Process Control pane visible, select **View** → **Process Control** to display it as a floating pane, shown in Figure 2-4.

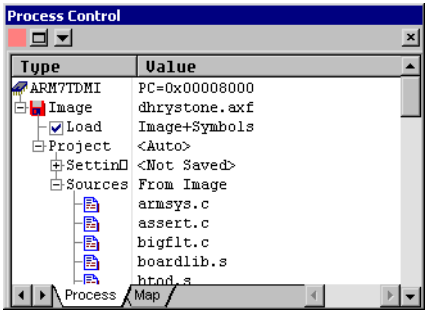

Figure 2-4 Image details in the Process Control pane

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* for details.

Map Displays the memory mapping for the target processor, or the current process. You can also temporarily change the map settings if required.

See Chapter 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 chapters describing RTOS support and multiprocessing in *RealView Debugger v1.8 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 loaded (that is, image with symbols, or symbols only)
- associated files.

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

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

In the example in Figure 2-4 on page 2-11, you can see the entries:

Current process

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

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

Image Details the loaded images:

Load For each image, a check box indicates the load state and what has been loaded, that is image, symbols, or both.

Project Shows that the project associated with the connection is either a real, user-defined project file (shown by the project name) or an auto-project (shown by `<Auto>`).

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

Sources These are either the sources making up the project, sources extracted from the makefile used in the build, or sources from the loaded image.

Depending on the type of project, right-click on this entry to display a context menu to specify how sources are collected.

Working with source files

When working with entries in the Process Control pane, you can use type ahead facilities to locate files. This is especially useful where your project specifies a large number of source files. For example, type the first letter of the source file that you want

to view. RealView Debugger expands the Sources entry and locates the first matching occurrence. When using this feature, the type ahead buffer is case insensitive and is limited to 128 characters. Do one of the following to clear the buffer:

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

Getting more information about an entry in the Process tab

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

Note

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

2.2.3 Working with auto-projects

An auto-project is an in-memory Custom project. The project settings are obtained from the image, where known. However, because no build details are known, the image cannot be rebuilt from the auto-project. See the *RealView Debugger v1.8 Project Management User Guide* for more details about auto-projects and Custom projects.

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

If, for example, you load the image dhrystone.axf, in *install_directory*\RVDS\Examples\...\dhrystone\DebugRel, RealView Debugger looks for the corresponding auto-project file dhrystone.axf.apr, in the same location. Where no auto-project exists, RealView Debugger creates an in-memory auto-project, named dhrystone. The **Process** tab is then updated with the project details, shown in Figure 2-5 on page 2-14.

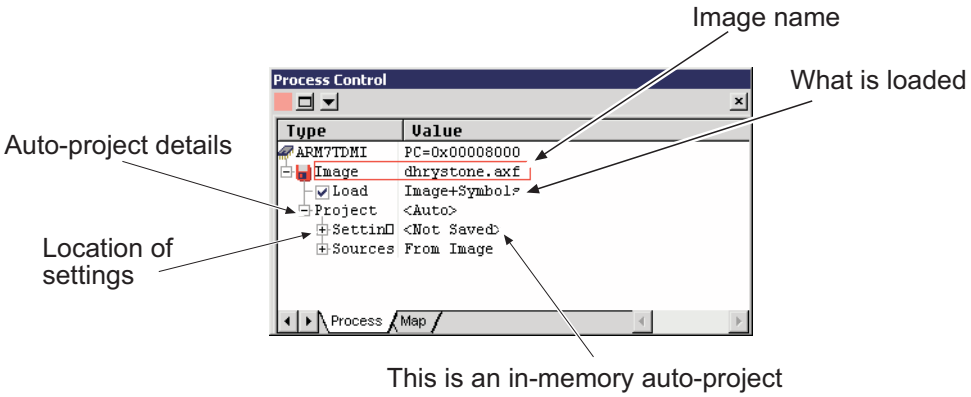

Figure 2-5 Auto-projects in the Process Control pane

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

Viewing project settings

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

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

Note

Do not leave the Project Properties window open when you are working with the debugger. When searching for source files, RealView Debugger updates the project properties as necessary. This fails if the window is already open (see *Searching for source files* on page 3-19 for details).

Changing project settings

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

Note

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

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

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

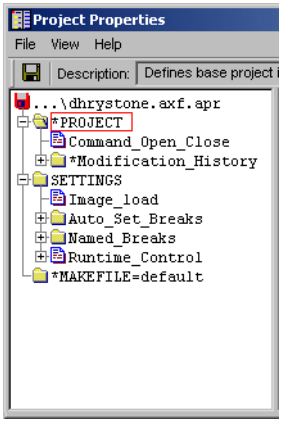

Figure 2-6 Changing auto-project settings

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

4. Expand the `SETTINGS` group to see the image settings.

   Figure 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. The changes are saved to the file `dhrystone.axf.apr`.

Note

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

Saving project settings from the Process Control pane

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

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

   Note

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

Deleting a saved auto-project file

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

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

Closing auto-projects

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

Note

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

See the chapter that describes working with multiple projects in *RealView Debugger v1.8 Project Management User Guide* 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.

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

See the description of the **Project** menu in the chapter that describes working with projects in *RealView Debugger v1.8 Project Management User Guide* for full details on these options.

Closing user-defined projects

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

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

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

If you do not unload the image, the debugger:

1. Closes the user-defined project.
2. Executes any close commands associated with the project.

3. Either:
   - opens the saved auto-project file, where one exists for the image
   - creates an in-memory project where no saved auto-project exists.

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

Note

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

See the chapter that describes working with projects in *RealView Debugger v1.8 Project Management User Guide* for more details on closing your projects.

2.3 Working with symbols

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

If you select the **Symbols Only** check box, on the Load File to Target dialog box (see Figure 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 or where you are working with an RTOS-enabled target.

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

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

Note

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

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 any of the load methods. This can be either a standalone image, or an image that is associated with a user-defined project.
2. Load a second image to the same target, for example `demo.axf`. The two images must not overlap in memory. You must load the second image as follows:
   a. Select **Target → Load Image...** from the main menu to display the Load File to Target dialog box (see *Using the Load File to Target dialog box* on page 2-3).
   b. Unselect the **Replace Existing File(s)** check box, to prevent this image replacing the first image.
   c. Unselect the **Auto-Set PC** check box, then unselect the **Set PC to Entry point** check box. This makes sure that the PC still points to the entry point in the first image.

      Note

      The **Set PC to Entry point** check box must be selected only for the image that contains the entry point, which might not be the first image you load.

   d. Click **Open**.
3. Select **View → Process Control** from the default Code window main menu to display the Process Control pane.
4. Expand the display to see the process details, shown in Figure 2-7.

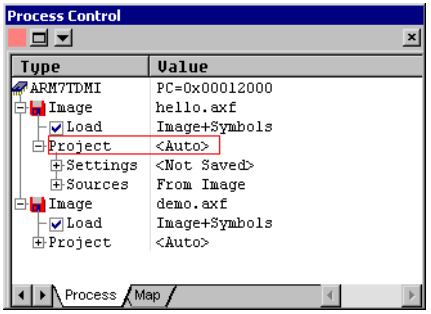

Figure 2-7 Multiple images in the Process Control pane

In this example, RealView Debugger:

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

For information on projects and project binding see *RealView Debugger v1.8 Project Management User Guide*.

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

- Chapter 3 *Controlling Execution* for details on setting scope.
- Chapter 6 *Working with Debug Views* for details on changing register entries.

Note

RealView Debugger can only keep track of the entry point for a single image. Therefore, if you are working with multiple images, you must set a manual breakpoint at the entry point of any other images you have loaded. This ensures that RealView Debugger is able to debug these images in the usual way.

See the chapter that describes multiprocessing in *RealView Debugger v1.8 Extensions User Guide* for more details on working with the Process Control pane in multiprocessor debugging mode.

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*
- *Replacing the currently loaded image* on page 2-23
- *Reloading an image* on page 2-24.

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 2-24.

To reset a processor:

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

Note

You cannot reset a target processor through a Multi-ICE connection using this method. You must use the Multi-ICE Server to perform a processor reset. See the *Multi-ICE User Guide* for more details.

2.5.2 Unloading an image

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

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

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

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

Unloading an image does not affect target memory. It unloads the symbol table (and any macros) and removes debug information from RealView Debugger. However, the image name is retained for reloading and the associated auto-project, or user-defined project used to load the image, is maintained.

Note

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

Deleting image details

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

Disconnecting with an image loaded

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

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

2.5.3 Replacing the currently loaded image

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

2.5.4 Reloading an image

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

- load the previously unloaded image
- reload the image.

Reloading refreshes any window displays and updates debugger resources.

To reload an image:

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

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

- Click the **Reload Image** button on the Image toolbar.

- Select the **Load** check box in the Process Control pane.

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

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

If you used the Load File to Target dialog box to enter a space-separated list of arguments to the image or comma-separated list of sections, these are not used when you reload (see *Using the Load File to Target dialog box* on page 2-3 for details). To reuse the arguments and sections either:

- Select **Target → Recent Images** to reload the image from the Recent Images list.

- Click the **Set PC to Entry** button on the Image toolbar to submit a `RESTART` command.

Chapter 3
Controlling Execution

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

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 Debug toolbar, or by typing a command-line instruction at the prompt.

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

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

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

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

The following notes apply to the command queue:

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

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

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 2-10, the File Editor pane contains three tabs:

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

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

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

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

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, that is the **Src** or **Dsm** tab. However, if you *force* scope to a different location then this is identified by a filled blue arrow and the red box highlights the current context. This might not be the true location of the PC. See *Forcing scope* for details.

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

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

Locating the Program Counter

You can locate the PC using the following methods:

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

Forcing scope

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

1. Connect to your target and load an image, for example `dhrystone.axf`.
2. Select **Edit → Advanced → Show Line Numbers** to display line numbers.

   This is not necessary but might help you to follow the examples.
3. Click on the **Src** tab to view the source file `dhry_1.c`.

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

4. Right-click at a location (line or address) in your execution view, for example line 149 in the source file `dhry_1.c`.

5. Select **Scope To Here** from the context menu.

   The forced scope is identified by a filled blue pointer at the chosen location. This moves the red box to highlight the current context.

6. Select **Debug → Show Context of PC** to see the location of the PC in the Output pane.

7. Click **High-level Step Into** to step into the program on the Debug toolbar.

8. Select **Debug → Show Context of PC** to see the location of the PC.

To reset the PC to the entry point, either:

- Select **Debug → Set PC to Entry Point** from the Code window main menu.

- Click **Set PC to Entry** on the Image toolbar.

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

Note

You can also scope to various parts of your application with the Data Navigator pane. See *Using the Data Navigator pane* on page 8-5 for details on how to use this pane.

Setting disassembly format

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

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

2. Click on the **Dsm** tab.

3. Right-click on whitespace inside the File Editor window to see the **Disassembly View** context menu.

4. Select **Set Disassembly Format...** from the context menu to see the Disassembly Mode selection box, shown in Figure 3-1 on page 3-6.

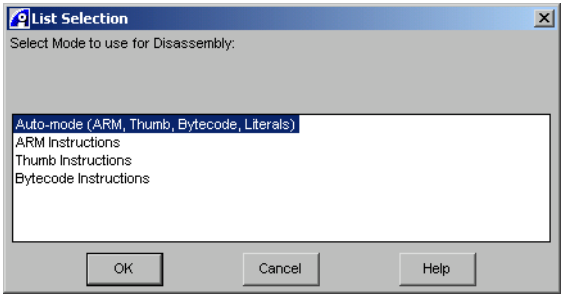

Figure 3-1 Disassembly Mode selection box

Use this selection box to specify how disassembled code is displayed, If you choose `Auto-mode`, the default format, RealView Debugger displays the code in a format specified by the image contents.

5. Select the required format and click OK to confirm your choice. Click **Cancel** to close the selection box without changing the display format.

You can use workspace settings to specify the format used for disassembly. For details, see the description of the `DEBUGGER` group (`Disassembler` settings) in Appendix A *Workspace Settings Reference*.

3.3 Using execution controls

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

- *Using the Execution group*
- *Using shortcuts* on page 3-9.

Note

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

3.3.1 Using the Execution group

The Execution group contains:

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

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

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

Stop Execution

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

If you are using semihosting with RealView ARMulator® ISS, you cannot use the **Stop Execution** button during the semihosting input.

High-level Step Into

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

To see an example using this option:

1. Connect to your target and load an image, for example `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Select **Edit → Advanced → Show Line Numbers** to display line numbers.

   This is not necessary but might help you to follow the examples.
4. Display line 78.
5. Set a default breakpoint by double-clicking on the line number in the left margin.
6. View the Output pane message:

   ```
   bi \DHRY_1\#78:2
   ```
7. Click **Run** and the program begins execution and runs up to the breakpoint.

   The Output pane shows where execution stops.
8. Click **High-level Step Into** once. A red box shows the location of the PC at line 91.
9. Click **High-level Step Into** several times to move through the lines of source code.

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

Note

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

High-level Step Over

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

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

Low-level Step Into

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

Low-level Step Over

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

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

Run until Return

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

Run to Cursor

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

Note

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

3.3.2 Using shortcuts

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

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

Table 3-1 Keyboard shortcuts

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

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, which provides a starting point for many debugging operations.

3.4.1 Using the Debug menu

The **Debug** menu offers the options:

Run With an image loaded to the target processor, select this option to run the program, starting from the current location of the PC. This issues the `GO` command (see *RealView Debugger v1.8 Command Line Reference Guide* for details).

Stop Execution

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

Set PC to Entry Point

Sets the PC to the entry point of the image. This issues the `RESTART` command (see *RealView Debugger v1.8 Command Line Reference Guide* for details). This means that you can run the image again, without having to do an image reload. Use this if you want to preserve:

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

Reset Target Processor

Performs a processor reset operation on the current connection. This issues the `RESET` command (see *RealView Debugger v1.8 Command Line Reference Guide* for details).

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

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

Note

This option is not supported for Multi-ICE connections. You must use the Multi-ICE Server to perform a processor reset. See the *Multi-ICE User Guide* for more details.

Step Into Steps execution, by lines of source code or assembler instructions, into functions. This behavior depends on the current code view, that is whether you have selected the **Src** tab or the **Dsm** tab. This issues the `STEP` command (see *RealView Debugger v1.8 Command Line Reference Guide* for details).

Step Over (next)

Steps execution, by lines of source code or assembler instructions, over functions. This behavior depends on the current code view, that is whether you have selected the **Src** tab or the **Dsm** tab. This issues the `STEPOVER` command (see *RealView Debugger v1.8 Command Line Reference Guide* for details).

Run until Return

Select this option to run from the current PC until control returns from the current function. This issues the `GO @1` command (see *RealView Debugger v1.8 Command Line Reference Guide* for details of the `GO` command).

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

Run to Cursor

With an image loaded, you can scroll down the source or disassembly listing and position the cursor on a specific line. Select **Run to Cursor** to execute the program up to the temporary breakpoint at the cursor, assuming that no halting condition occurs first. This issues either a `GO` *`source_line`* or `GO` *`address`* command depending on the current code view (see *RealView Debugger v1.8 Command Line Reference Guide* for details of the `GO` command).

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

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

Step Until Condition...

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

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

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

Breakpoints

This provides access to the **Breakpoints** menu to use breakpoints in your code. You can set unconditional software breakpoints, conditional software breakpoints from the **Conditional** submenu, and hardware breakpoints from the **Hardware** submenu. You must use a debug target that supports hardware breakpoints. See Chapter 4 *Working with Breakpoints* for more details on using these menu options.

Tracepoints

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

See the chapter that describes tracing in *RealView Debugger v1.8 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. See Chapter 7 *Reading and Writing Memory, Registers, and Flash* for more details on using these menu options.

Show Line at PC

Select this option to report the current module and procedure scope. This issues the `SCOPE` command (see *RealView Debugger v1.8 Command Line Reference Guide* for details).

Show Context of PC

Select this option to report the current context showing the current root, module, procedure, and line. This issues the `CONTEXT` command (see *RealView Debugger v1.8 Command Line Reference Guide* for details).

Toggle Source/Disassembly

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

Set Source Search Path...

Use this to display the Project Properties window, where you can set the path to enable RealView Debugger to find sources where the compiler or assembler does not pass source paths (see *Searching for source files* on page 3-19 for details).

3.5 Automating debugging operations

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

- *Creating log and journal files*
- *Using include files* on page 3-18.

3.5.1 Creating log and journal files

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

Types of log and journal files

RealView Debugger enables you to create the following files:

Log files During your debugging session, you can create a log file containing:

- all the commands you enter
- commands that are generated by the RealView Debugger GUI.

You can then use this file as the basis of a command file or a macro. By default, log files have the extension `.log` or `.inc`, but you can use any extension for writing.

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

Journal files A session journal file that contains all information including:

- commands you enter
- commands that are generated by the RealView Debugger GUI
- any messages displayed in the Output pane
- messages from your application.

By default, journal files have the extension `.jou`, but you can use any extension for writing.

Creating a log file for use as an include file

To create an log file for use as an include file, or to append to an existing log file:

1. Select **Tools → Logs and Journal → Log File...** to display the Select File to Log to dialog box where the file can be located.
2. Specify the pathname of the new log file, or locate a file created previously, for example `\home\my_user_name\my_log.log`.
3. Click **Save** to confirm the settings and close the dialog box.
4. If the specified log file already exists, RealView Debugger displays the File Exists prompt.

   This gives you the option to append or replace. Click:

   - **Yes** to append new commands to those already saved in the file
   - **No** to replace, or overwrite, any commands already saved in the file
   - **Cancel** to close the prompt and discontinue the log file access.

Output is now recorded automatically in the specified file, unless you choose to cancel logging.

RealView Debugger shows that it is recording using the status display area at the bottom of the Code window, shown in Figure 3-2.

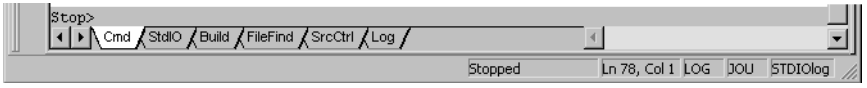

Figure 3-2 Output files recording

Closing log and journal files

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

To close a log or journal file, select **Tools → Logs and Journal → Close Logs/Journals...**. This displays a list selection box, shown in Figure 3-3 on page 3-17, where you can specify which file, or files, to close.

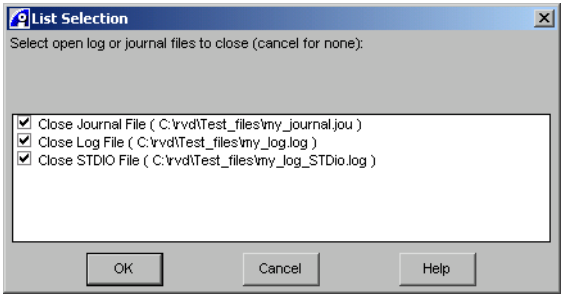

Figure 3-3 Close log and journal files selection box

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

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

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

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

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

Creating log and journal files at start-up

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

```
rvdebug.exe -stdiolog "C:\rvd\test_files\STDIO_tst_file.log"
rvdebug.exe -jou "C:\rvd\test_files\Jou_tst_file.jou"
rvdebug.exe -log "C:\rvd\test_files\Log_tst_file.log"
```

If the file does not exist, RealView Debugger creates it. Where the file exists, RealView Debugger overwrites the current contents, without displaying a warning message.

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

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

3.5.2 Using include files

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

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

- *Output buffering*
- *Including files*.

Output buffering

In the current release of RealView Debugger, output to the log files and journal files is buffered. This means that all lines are not immediately flushed to the specified file. To change this, so that output to a file is unbuffered, set the `JOULOG_UNBUF` environment variable to any value.

Including files

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

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

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

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

3.6 Searching for source files

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

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

This section includes:

- *Example of locating source files*
- *Autoconfiguring search rules* on page 3-20
- *Manually configuring search rules* on page 3-21
- *Source path mappings* on page 3-22.

3.6.1 Example of locating source files

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

1. Ensure that the Project Properties window is not open. When searching for source files, RealView Debugger updates the project properties as necessary. This fails if the window is already open.
2. Close any files that are open in the File Editor so that they are not found, and included, in the search mechanism.
3. Connect to your target and load an image, for example `dhry_thr_demo.axf`.

   RealView Debugger creates an in-memory auto-project to use for this session, that is `dhry_thr_demo.axf.apr`.
4. RealView Debugger cannot locate a source file and so displays the source search prompt, shown in Figure 3-4 on page 3-20.

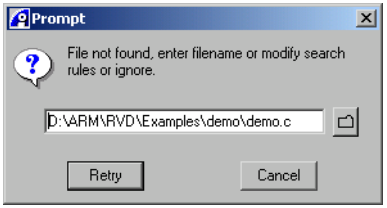

Figure 3-4 Source search prompt

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

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

- *Autoconfiguring search rules*
- *Manually configuring search rules* on page 3-21.

3.6.2 Autoconfiguring search rules

Use the source search prompt, shown in Figure 3-4, to autoconfigure the search rules:

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

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

Note

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

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

Note

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

3.6.3 Manually configuring search rules

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

To configure the search rules manually:

1. Select **Project → Project Properties...** to display the Project Properties window, shown in Figure 3-5.

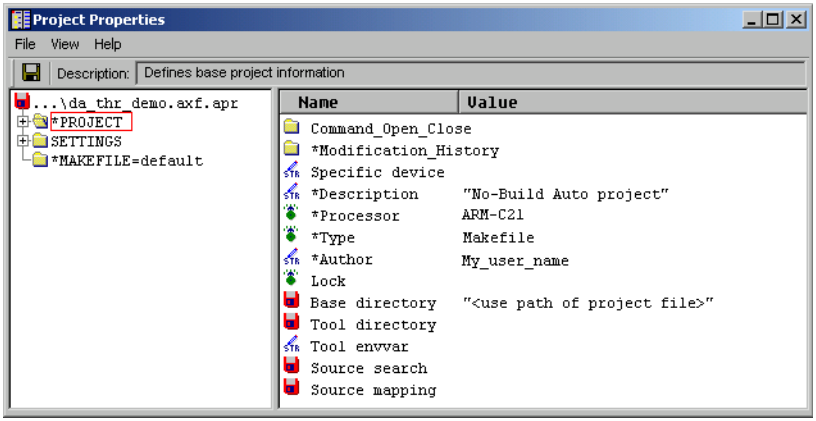

Figure 3-5 Search rules in the Project Properties window

   For more details on other entries in this window see *Source path mappings* on page 3-22.

2. Right-click on `Source_search` and use **Edit as Directory Name...** to specify the search path to locate the missing source file.

3. Click **File → Save and Close** to confirm your choice and close the Project Properties window.

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

Note

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

3.6.4 Source path mappings

When you use the directory button to locate the source file, using the prompt shown in Figure 3-4 on page 3-20, RealView Debugger creates a mapping between the original file and the new location. This mapping is then applied to subsequent file searches so that the debugger can locate files automatically that have the same mapping.

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

Note

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

See the chapter that describes customizing projects in *RealView Debugger v1.8 Project Management User Guide* for full details on working with these entries.

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:

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
- output messages
- update windows or files
- change the behavior of your application program.

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

This section describes:

- *Breakpoint types*
- *Qualifying breakpoint line number references with module names* on page 4-7
- *Specifying address ranges* on page 4-7
- *Specifying the entry point to a function* on page 4-8
- *Breakpoints and memory map locations* on page 4-9
- *Viewing breakpoints in your code view* on page 4-10
- *Halted System Debug and Running System Debug* on page 4-12
- *Using hardware breakpoints* on page 4-13
- *Breakpoints and image restarts* on page 4-14
- *Using breakpoints with RealView ICE* on page 4-14.

4.1.1 Breakpoint types

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

- *Software and hardware breakpoints* on page 4-3
- *Unconditional and conditional breakpoints* on page 4-4
- *Default breakpoints in the GUI* on page 4-5

- *Recording the triggering of a breakpoint* on page 4-6.

The type of breakpoint you can set depends on the:

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

RealView Debugger uses breakpoints to temporarily halt execution as you step through your application. For more details about stepping through code, see *Using execution controls* on page 3-7.

Software and hardware breakpoints

RealView Debugger enables you to set software or hardware breakpoints, depending on the target and the chosen memory location (see *Breakpoints and memory map locations* on page 4-9):

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

You can set as many software breakpoints as you require.

Note

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

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

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

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

Note

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

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

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

Check your hardware characteristics, and your vendor-supplied documentation, to determine the level of support available for hardware breakpoints. Also, see *Viewing your hardware breakpoint support* on page 4-13.

Note

For RVISS, only the RealView Connection Broker interface supports hardware breakpoints.

Unconditional and conditional breakpoints

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

Unconditional

A breakpoint is unconditional when no condition qualifier is assigned to the breakpoint. However, you can:

- Assign one or more actions to an unconditional breakpoint. When the breakpoint is triggered, the specified action is performed (see *Setting unconditional breakpoints in other ways* on page 4-22).
- Choose whether it is a software or hardware breakpoint, depending on the memory region (see *Breakpoints and memory map locations* on page 4-9).

For an unconditional breakpoint, the program always stops if execution reaches a specified location.

Default breakpoints

You can set an unconditional breakpoint that has no actions assigned to it, and without explicitly choosing whether it is a software or hardware breakpoint. This is called a *default breakpoint*, and either RealView Debugger or your target

hardware determines whether a software or hardware breakpoint is set (see *Setting default breakpoints* on page 4-15).

Conditional A breakpoint is conditional when a condition qualifier is assigned to the breakpoint. The conditions that must be satisfied can be defined based on data values, the result of a macro, or on counters.

The program stops if execution reaches a specified location and one or more of the predefined conditions are met (see *Setting conditional breakpoints* on page 4-34).

If you want to use multiple conditions, see *Controlling the behavior of breakpoints* on page 4-59 for details on how the order of conditional qualifiers affects the behavior of a breakpoint.

Note

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

Default breakpoints in the GUI

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

A default breakpoint does not have any condition qualifiers or actions, and so is always an unconditional breakpoint. However, the chosen memory location where you set the breakpoint determines whether it is a software or hardware breakpoint (see *Breakpoints and memory map locations* on page 4-9).

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

- To set a default software breakpoint, RealView Debugger uses the `BREAKINSTRUCTION` command. For example, to set a default breakpoint at the RAM address `0x00008008`, RealView Debugger issues the command:

  ```
  binstr (I A)0x00008008
  ```

- To set a default hardware breakpoint, RealView Debugger uses the `BREAKEXECUTION` command. For example, if you set a default breakpoint at the Integrator™/AP Flash address `0x24000008`, RealView Debugger issues the command:

```
bexec (I A)0x24000008
```

Note

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

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

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

For more details about these commands, see the *RealView Debugger v1.8 Command Line Reference Guide*.

For detailed instructions on setting default breakpoints, see *Setting default breakpoints* on page 4-15.

Note

Default breakpoints are not added to the breakpoint history (see *Setting breakpoints from saved lists* on page 4-74).

Recording the triggering of a breakpoint

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

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

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

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

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

See *Controlling the behavior of breakpoints* on page 4-59 for more details.

4.1.2 Qualifying breakpoint line number references with module names

If you want to set a breakpoint in a source file that contains the line of code pointed to by the PC, you can specify only a line number in the file. For example, using the `BREAKINSTRUCTION` command you can set a breakpoint at column 2 of line 92 in the `dhry_1.c` source file with:

```
breakinstruction #92:2
```

However, if you want to set a breakpoint in another source file associated with the loaded image, you must qualify the line number reference with the module name.

Module naming conventions

For sources with the `.c` or `.cpp` extension, the module name is the source filename without the extension. If the extension is not `.c` or `.cpp`, the extension is preserved, and the dot is replaced with an underscore. All module names are converted to uppercase by RealView Debugger. Therefore, you must specify module names in uppercase. For example, `sample_arm.c` is converted to `SAMPLE_ARM`, and `sample_arm.s` is converted to `SAMPLE_ARM_S`.

If two modules have the same name then RealView Debugger appends an underscore followed by a number to the second module, for example `SAMPLE_ARM_1`. Additional modules are numbered sequentially, for example, if there is a third module this becomes `SAMPLE_ARM_2`.

Following this convention avoids any confusion with the C dot operator indicating a structure reference.

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

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

4.1.3 Specifying address ranges

Hardware breakpoints enable you to specify a range of addresses. You specify an address range using either of the following formats:

`start_addr..end_addr`

Start address and an absolute end address, for example:

```
0x10000..0x20000
```

`start_addr..+length`

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

```
0x10000..+0x10000
```

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

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

```
main..+1000 Arr_2_Glob..+64
```

4.1.4 Specifying the entry point to a function

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

To set a breakpoint on the entry to a function, specify *`function`*`\@entry`. This identifies the first location in the function after the code that sets up the function parameters. In general, *`function`*`\@entry` refers to either:

- the first executable line of code in that function
- the first auto local that is initialized in that function.

If no lines exist that set up any parameters for the function (for example, an embedded assembler function), then the following error message is displayed:

```
Error: E0039: Line number not found.
```

As an example, if you have a function `func_1(value)` you might want to set a breakpoint that triggers only when the argument `value` has a specific value on entry to the function:

```
bi,when:{value==2} func_1\@entry
```

Note

This is different to specifying a function name without the `@entry` qualifier. For example, in the `dhrystone.axf` example image:

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

4.1.5 Breakpoints and memory map locations

With memory mapping enabled, RealView Debugger sets a breakpoint based on the access rule for the memory at the chosen location. Table 4-1 shows the different types of location that you can define in a memory map with RealView Debugger, and the type of breakpoint that RealView Debugger sets by default.

Table 4-1 Breakpoint types for different memory map locations

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

If you have to, you can set a hardware breakpoint in RAM. To do this, you must use one of the methods described in *Setting hardware breakpoints explicitly* on page 4-24.

RealView Debugger cannot determine the type of default breakpoint to set in WOM memory, and displays the Set Address/Data Breakpoint dialog box (see *Displaying the Set Address/Data Breakpoint dialog box* on page 4-41). This dialog box is also displayed if an error occurs when you attempt to set a breakpoint.

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

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

For more details about memory mapping, see Chapter 5 *Memory Mapping*.

4.1.6 Viewing breakpoints in your code view

Breakpoints are marked in the source-level and disassembly-level view in the gray margin, at the left side of the window. Standard software and hardware breakpoints are identified by a disc icon, which is color-coded to reflect the type (see *Breakpoint types* on page 4-2) and status of the breakpoint:

Red A red icon shows that you have set an unconditional breakpoint, and that the breakpoint is enabled.

Yellow A yellow icon shows that you have set a conditional breakpoint, and that the breakpoint is enabled. The conditions are software conditions that are controlled by RealView Debugger. Although hardware breakpoints can have conditions that are controlled in hardware, only when the software conditions are assigned are these breakpoints conditional in RealView Debugger.

White A white icon shows that you have disabled an existing breakpoint. When you re-enable the breakpoint, the icon changes back to the color it had before the breakpoint was disabled.

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

Note

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

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

Multiple breakpoints units on a single line

If you have set multiple breakpoint units on a source line containing multiple statements or on inline functions, then disabling the first breakpoint unit changes the marker disc, shown in Figure 4-1 on page 4-11. If you disable the second breakpoint unit on the line, the marker remains.

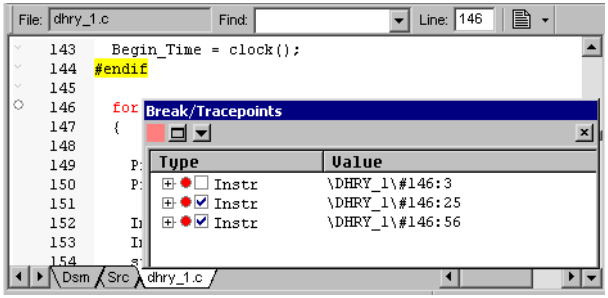

Figure 4-1 Working with multiple breakpoint units

See *Using the Break/Tracepoints pane* on page 4-63 for details on viewing breakpoints in the Break/Tracepoints pane.

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

Clearing breakpoints quickly

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

Note

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

Other types of breakpoints

If you have set thread breakpoints, these are shown as different types of marker using icons as shown in Table 4-2. See *Halted System Debug and Running System Debug* on page 4-12, and *RealView Debugger v1.8 Extensions User Guide* for details of using these features.

Table 4-2 Other breakpoint/tracepoint icons

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

4.1.7 Halted System Debug and Running System Debug

RealView Debugger supports different debugging modes:

Halted System Debug

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

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

Running System Debug

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

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

RSD gives access to the application using a *Debug Agent* (DA) that resides on the target and is typically considered to be part of the RTOS. The Debug Agent is scheduled along with other tasks in the system.

Where RSD is supported, RealView Debugger enables you to switch seamlessly between RSD and HSD mode using GUI controls or CLI commands.

For full details on using RTOS extensions and running in RSD mode, see the chapter that describes RTOS support in *RealView Debugger v1.8 Extensions User Guide*.

4.1.8 Using hardware breakpoints

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

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

Be aware of the following when working with hardware breakpoints with *RealView ARMulator® ISS* (RVISS):

- Watchpoints are available. These are called hardware breakpoints in RealView Debugger. You can access them through the **Debug → Breakpoints → Hardware** menu. These data access breakpoints are implemented using a memory hook.
- Hardware breakpoints can use address ranges (see *Specifying address ranges* on page 4-7), data values, and data value range tests. They can also include size tests, mode tests, and pass counts.
- Hardware breakpoints can be chained to form complex tests.

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

Viewing your hardware breakpoint support

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

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

This displays an information box describing the support available for your target processor, shown in Figure 4-2.

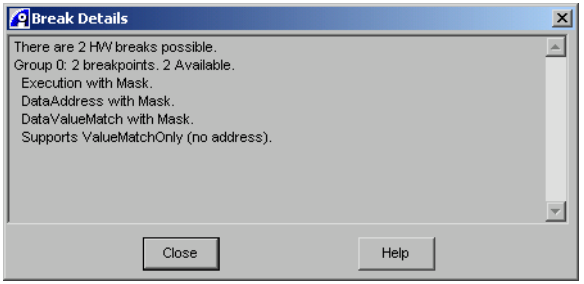

Figure 4-2 Example hardware break characteristics

What is shown in this details display depends on the target processor and the target vehicle used to make the connection. Figure 4-2 shows the details for an ARM940T using Multi-ICE®.

Note

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

4.1.9 Breakpoints and image restarts

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

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

However, if you have set any breakpoints or tracepoints, the first method causes them to be cleared. If you want to preserve any existing breakpoints and tracepoints, then use either of the other methods. See Chapter 3 *Controlling Execution* for more details.

4.1.10 Using breakpoints with RealView ICE

If you are using RealView ICE, RealView Debugger sets breakpoints differently. For full details on debugging with RealView ICE, see *RealView ICE User Guide*.

4.2 Setting default breakpoints

You can set a default breakpoint directly from the Code window (see *Default breakpoints in the GUI* on page 4-5). The methods you can use to do this depend on your current code view. A default breakpoint is a useful way to set a quick test point during a debugging session.

The type of default breakpoint set, either software or hardware, is determined by the chosen memory location (see *Breakpoints and memory map locations* on page 4-9).

Note

Default breakpoints are not added to the breakpoint history (see *Setting breakpoints from saved lists* on page 4-74).

This section describes:

- *Default breakpoints in source-level view* on page 4-16
- *Default breakpoints in disassembly-level view* on page 4-17
- *Setting breakpoints on branch instructions* on page 4-18
- *Setting default breakpoints in other ways* on page 4-19.

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

Edit → Advanced → Show Line Numbers

Note

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

4.2.1 Default breakpoints in source-level view

Double-click in the margin, that is the gray area to the left of a line, to set a default breakpoint quickly, shown in Figure 4-3. You can also double-click on the line number if this is visible.

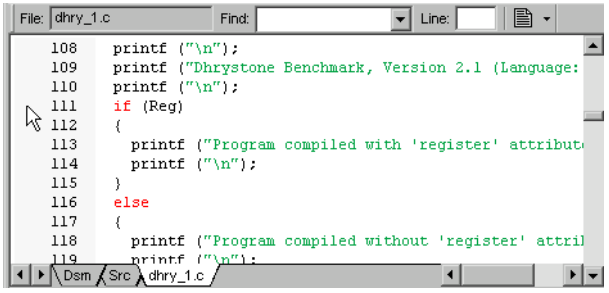

Figure 4-3 Setting an unconditional breakpoint on a line

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

Setting a breakpoint on a multi-statement line

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

Setting a breakpoint on a symbol

If you want to set a breakpoint on a symbol, right-click on the symbol (for example, a function name) to display the **Symbol** context menu, and select **Set Break On**. You can view the currently defined symbols with the `PRINTSYMBOL` command, described in the *RealView Debugger v1.8 Command Line Reference Guide*.

Note

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

Setting a breakpoint in the body of a function

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

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

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

Effects of setting a breakpoint

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

```
bi \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 in RAM.

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

See *Using the Break/Tracepoints pane* on page 4-63 for details on viewing breakpoints in the Break/Tracepoints pane.

Note

For details of the `BREAKINSTRUCTION` and `BREAKEXECUTION` commands see *RealView Debugger v1.8 Command Line Reference Guide*.

4.2.2 Default breakpoints in disassembly-level view

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

4.2.3 Setting breakpoints on branch instructions

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

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

   Note

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

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

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

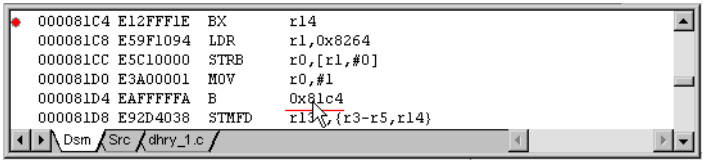

Figure 4-4 Setting an unconditional breakpoint on an instruction

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

   ```
   bi 0x81c4
   ```

Note

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

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

See *Using the Break/Tracepoints pane* on page 4-63 for details on viewing breakpoints in the Break/Tracepoints pane.

4.2.4 Setting default breakpoints in other ways

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

- Use one of the generic breakpoint operations described in *Generic breakpoint operations* on page 4-20.
- Use drag-and-drop to create a breakpoint in the Break/Tracepoints pane based on a program item. For example, highlight a source code function in the File Editor pane and then drag it (using your mouse) and drop it into the Break/Tracepoints pane (see *Using the Break/Tracepoints pane* on page 4-63 for details).
- Right-click in the left margin of the File Editor pane, and select **Set Break** from the context menu.
- Right-click on a function in the **Functions** tab of the Data Navigator pane, and select **Break** from the context menu. See *Using the Data Navigator pane* on page 8-5 for more details on working with the Data Navigator pane.

4.3 Generic breakpoint operations

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

Toggle Break at Cursor

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

- If no breakpoint exists at the address, a new default breakpoint is created (see *Default breakpoints in the GUI* on page 4-5).
- If a breakpoint already exists at the address, it is cleared.

Enable/Disable Break at Cursor

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

- If an enabled breakpoint already exists at the address, it is disabled.
- If a disabled breakpoint already exists at the address, it is enabled.
- If no breakpoint exists at the address, a new default breakpoint is created (see *Default breakpoints in the GUI* on page 4-5).

Clear All Break/Tracepoints

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

See *Disabling and clearing breakpoints* on page 4-72 for more details of disabling and clearing breakpoints.

4.4 Setting unconditional breakpoints explicitly

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

- use the Simple Break if X dialog box (see *Using the Simple Break if X dialog box*)
- use the Set Address/Data Breakpoint dialog box without assigning any qualifiers (see *Using the Set Address/Data Breakpoint dialog box* on page 4-41).

4.4.1 Using the Simple Break if X dialog box

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

To display this dialog box, shown in Figure 4-5, use one of the following methods:

- Select the following option from the Code window main menu:

  Debug → Breakpoints → Conditional → Break if X...

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

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

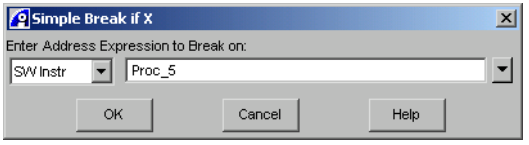

Figure 4-5 Simple Break if X dialog box

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

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

Specify the location to break on. This can be:

- a specific line number in the source code, with or without a module name prefix (see *Qualifying breakpoint line number references with module names* on page 4-7)
- a specific address or address range (see *Specifying address ranges* on page 4-7)
- a function entry (see *Specifying the entry point to a function* on page 4-8).

You can enter a location directly in the field, or click the drop-down arrow to the right of the field to choose a location from a list browser (see *List browser dialog boxes* on page 8-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 8-32), or select from a list of previously-used expressions.

4.4.2 Setting unconditional breakpoints in other ways

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

- *Setting a breakpoint at a memory location*
- *Setting a breakpoint at the address of a symbol*
- *Setting a breakpoint on a chosen function or variable* on page 4-23
- *Assigning an action to an unconditional breakpoint* on page 4-23.

Setting a breakpoint at a memory location

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

See the *Operating on memory contents* on page 6-25 for details.

Setting a breakpoint at the address of a symbol

Set a hardware breakpoint at an address of a symbol using the **Set Break At...** option on the Stack pane context menu, but do not assign a qualifier to the breakpoint. This displays the Set Address/Data Breakpoint dialog box (see *Using the Set Address/Data Breakpoint dialog box* on page 4-41).

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.

Setting a breakpoint on a chosen function or variable

You can use the Data Navigator pane to set an unconditional breakpoint on a chosen function or variable, defined as a location in the image. See *Using the Data Navigator pane* on page 8-5 for details.

Assigning an action to an unconditional breakpoint

You can assign one or more actions to an unconditional breakpoint. To do this, you must use the Set Address/Data Breakpoint dialog box to create a new unconditional breakpoint or to edit an existing unconditional breakpoint (see *Using the Set Address/Data Breakpoint dialog box* on page 4-41).

4.5 Setting hardware breakpoints explicitly

Use hardware breakpoints to set execution or data breaks, or where your code is stored in ROM or Flash (see *Breakpoints and memory map locations* on page 4-9). The facilities available depend on the current debug target, that is both the target processor and the target vehicle.

To set hardware breakpoints, you must be connected to a debug target that supports these features, such as an ARM processor with EmbeddedICE logic (for example, an ARM966E-S), or simulator software that supports this feature (for example, RealView Simulator Broker connections to RVISS).

Menu options related to hardware breakpoints are grayed out if your target cannot support them. Similarly, RealView Debugger displays a message if you select an option from a drop-down list box that is not supported by your debug target. Check your hardware characteristics (see *Viewing your hardware breakpoint support* on page 4-13), and your vendor-supplied documentation, to determine the level of support for hardware breakpoints.

This section includes:

- *Using the HW Break if in Range dialog box*
- *Using the HW While in function/range, Break if X dialog box* on page 4-26
- *Using the HW Break if X, then if Y dialog box* on page 4-27
- *Using the HW Break on Data Value match dialog box* on page 4-29
- *Chaining breakpoints* on page 4-30.

Note

The methods described in this section create unconditional hardware breakpoints. To set conditional hardware breakpoints, see *Setting conditional hardware breakpoints* on page 4-38.

4.5.1 Using the HW Break if in Range dialog box

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

Displaying the dialog box

To display the HW Break if in Range dialog box, shown in Figure 4-6 on page 4-25, select the following option from the Code window main menu:

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

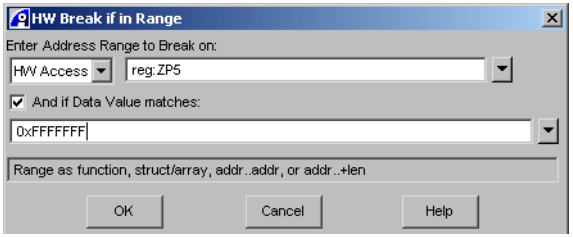

Figure 4-6 HW Break if in Range dialog box

HW Break if in Range dialog box interface components

The HW Break if in Range dialog box (see Figure 4-6) contains the following fields:

Enter Address Range to Break on

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

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

Specify the location to break on. This can be:

- a specific line number in the source code, with or without a module name prefix (see *Qualifying breakpoint line number references with module names* on page 4-7)
- a specific address or address range (see *Specifying address ranges* on page 4-7)
- a function entry (see *Specifying the entry point to a function* on page 4-8).

And if Data Value matches

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

You can enter values directly in these fields, or click the drop-down arrow to the right of these fields to choose from a list browser (see *List browser dialog boxes* on page 8-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 8-32), or select from a list of previously-used expressions.

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

The HW While in function/range, Break if X dialog box enables you to specify a hardware breakpoint that uses two chained breakpoint units (see *Chaining breakpoints* on page 4-30 for more information).

Displaying the dialog box

To display the HW While in function/range, Break if X dialog box, shown in Figure 4-7, select the following option from the Code window main menu:

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

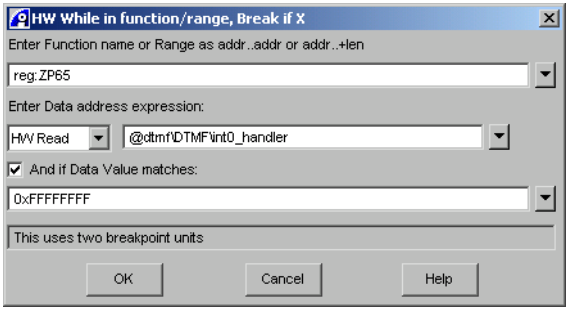

Figure 4-7 HW While in func/range, Break if X dialog box

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

The HW While in function/range, Break if X dialog box (see Figure 4-7) contains the following fields:

Enter Function name or Range

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

- a specific line number in the source code, with or without a module name prefix (see *Qualifying breakpoint line number references with module names* on page 4-7)
- a specific address or address range (see *Specifying address ranges* on page 4-7)
- a function entry (see *Specifying the entry point to a function* on page 4-8).

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

Enter Data address expression

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

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

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

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

And if Data Value matches

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

You can enter values directly in these fields, or click the drop-down arrow to the right of these fields to choose from a list browser (see *List browser dialog boxes* on page 8-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 8-32), or select from a list of previously-used expressions.

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

The HW Break if X, then if Y dialog box enables you to specify a hardware breakpoint that uses two chained breakpoint units (see *Chaining breakpoints* on page 4-30 for more information). The breakpoint is set at the specified memory location or address range (see *Specifying address ranges* on page 4-7) depending on the values read.

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

To display the HW Break if X, then if Y dialog box, shown in Figure 4-8, select the following option from the Code window main menu:

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

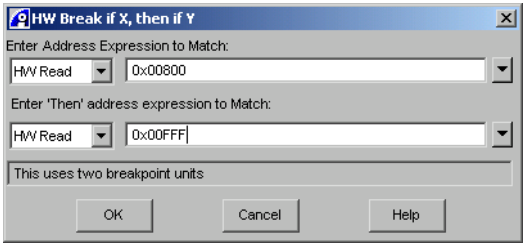

Figure 4-8 HW break if X, then if Y dialog box

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

The HW Break if X, then if Y dialog box (see Figure 4-8 on page 4-27) contains the following fields:

Enter Address Expression to Match

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

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

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

Enter 'Then' address expression to Match

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

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

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

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

- a specific line number in the source code, with or without a module name prefix (see *Qualifying breakpoint line number references with module names* on page 4-7)
- a specific address or address range (see *Specifying address ranges* on page 4-7)
- a function entry (see *Specifying the entry point to a function* on page 4-8).

You can enter the location directly into these fields, or click the drop-down arrow to the right of these fields to choose a location from a list browser (see *List browser dialog boxes* on page 8-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 8-32), or select from a list of previously-used expressions.

4.5.4 Using the HW Break on Data Value match dialog box

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

Displaying the HW Break on Data Value match dialog box

To display the HW Break on Data Value match dialog box, shown in Figure 4-9, use one of the following methods:

- Select the following option from the Code window main menu:

  Debug → Breakpoints → Hardware → HW Break on Data Value match...

- Using the Break/Tracepoints pane, either:
  - — select **Set BreakIf...** from the **Pane** menu
  - — right-click on the pane background, and select **Set BreakIf...** from the context menu.

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

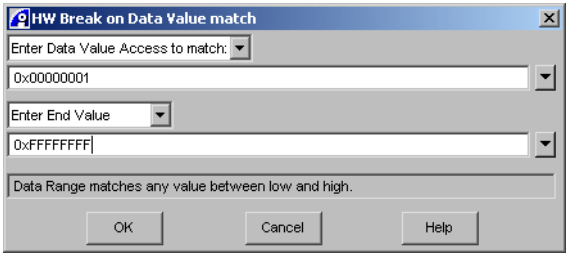

Figure 4-9 HW Break on Data Value match dialog box

HW Break on Data Value match dialog box interface components

The HW Break on Data Value match dialog box (see Figure 4-9) contains the following fields:

Data value to match

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

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

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

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

Data value modifier

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

Enter None (no range)

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

Enter End Value

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

Enter Value Length

Test for values in the range from the first data value and the first data value plus the length specified, inclusive. For example, if the first data value is `0x10`, and the length is `0xF`, then the range is from `0x10` to `0x1F`, inclusive.

Enter Value Mask

Specify a value mask if you are interested only in certain bits of the value. For example, if you have a data field occupying five bits (`0x1F`), then set a mask of `0x5` if only bits 0 and 2 are of interest.

You can enter values directly in these fields, or click the drop-down arrow to the right of these fields to choose from a list browser (see *List browser dialog boxes* on page 8-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 8-32), or select from a list of previously-used expressions.

4.5.5 Chaining breakpoints

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

- Using the following dialog boxes:
  - HW While in function/range, Break if X dialog box (see *Using the HW While in function/range, Break if X dialog box* on page 4-26)
  - HW Break if X, then if Y dialog box (see *Using the HW Break if X, then if Y dialog box* on page 4-27).
- Using break command qualifiers, see:
  - *Chaining breakpoints with command qualifiers* on page 4-31

— *Converting existing individual breakpoints to chained breakpoints*.

Chaining breakpoints with command qualifiers

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

For more details on the break commands, see the *RealView Debugger v1.8 Command Line Reference Guide.*

Converting existing individual breakpoints to chained breakpoints

If you have created individual breakpoints, and you want to chain them together, you must use CLI commands to create new breakpoints based on the existing breakpoints. You must create new breakpoints because you cannot use the `modify` qualifier with the `hw_and` qualifier. However, you can modify a chained breakpoint with any other qualifier and also change the address expression.

To create chained breakpoints based on existing breakpoints:

1. Display the Break/Tracepoints pane.
2. Expand the breakpoint instances to determine the CLI command that was used to create each breakpoint.
3. Make a note of the commands used to create each breakpoint. Alternatively:
   - copy the commands from the output in the **Cmd** tab, and save them in a text file
   - copy the commands from the log or journal file, if you have one.
4. Clear the original breakpoints that you are using to create the chained breakpoints.
5. Determine the order in which the breakpoints are to be chained.
6. For each breakpoint command you enter, include the `hw_and` qualifier to set up the chained breakpoints as follows:
   a. For the first breakpoint in the chain, include the `hw_and:next` or `hw_and:"then-next"` command qualifiers.
   b. For each subsequent breakpoint in the chain, include the `hw_and:prev` or `hw_and:"then-prev"` command qualifiers.

For more details on the break commands, see the *RealView Debugger v1.8 Command Line Reference Guide.*

4.6 Specifying processor exceptions (global breakpoints)

If supported by your debug target, RealView Debugger maintains a list of processor exceptions that automatically trigger a breakpoint in any application program. These are global breakpoints, that is they apply to processor exceptions and not addresses. During your debugging session you can examine this list and select, or deselect, events that halt the processor.

4.6.1 Displaying the List Selection dialog box

To display the List Selection dialog box, shown in Figure 4-10, use one of the following methods:

- Select the following option from the Code window main menu:

  Debug → Processor Exceptions...

- Using the Break/Tracepoints pane, either:
  - select **Processor Exceptions...** from the **Pane** menu
  - right-click on the pane background, and select **Processor Exceptions...** from the context menu.

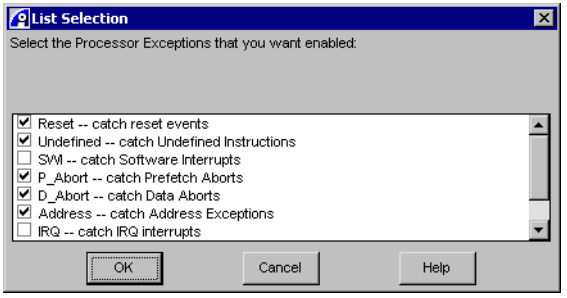

Figure 4-10 Processor Exceptions List Selection dialog box

4.6.2 List Selection dialog box interface components

The List Selection dialog box (see Figure 4-10) shows processor exceptions that stop execution. An exception is enabled when the associated check box contains a tick:

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

4.7 Setting conditional breakpoints

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

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

This section includes:

4.7.1 Considerations when using conditional breakpoints

Conditional breakpoints can be very intrusive because RealView Debugger takes control when the breakpoint triggers. The specified qualifiers are checked and then, if applicable, control is returned to the application. See *Controlling the behavior of breakpoints* on page 4-59 for details on how the order of conditional qualifiers affects the behavior of a breakpoint.

When a conditional breakpoint triggers, there might be a discrepancy between the real state of your target and what is shown in the Code window. For example, the State field might show that the target has stopped when it is running. This is because the state of the target, as reported by RealView Debugger, is a snapshot of the target status at the time that the breakpoint triggered. Depending on the tasks on the system, the target state might have changed. Therefore, when you are using conditional breakpoints, remember that the state of the target depends on several factors, including:

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

Note

If you are using RTOS extensions and running in RSD mode, see the chapter that describes RTOS support in *RealView Debugger v1.8 Extensions User Guide* for more details on working with breakpoints.

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

The Simple Break if X, N times dialog box, shown in Figure 4-11, enables you to set a conditional breakpoint that specifies how many times execution must arrive at the specified address before the breakpoint triggers. This dialog box provides a quick way of assigning the **SW Pass Count** qualifier to a breakpoint (see *Specifying Qualifiers* on page 4-52).

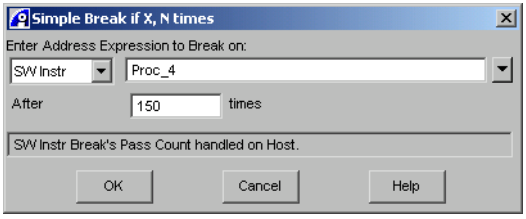

Figure 4-11 Simple Break if X, N times dialog box

Displaying the Simple Break if X, N times dialog box

To display the Simple Break if X, N times dialog box (see Figure 4-11) use one of the following methods:

- Select the following option from the Code window main menu:

  Debug → Breakpoints → Conditional → Break if X, N times...

- Right-click on the margin, an instruction, or line of source code in the File Editor pane, and select **Set BreakIf...** from the context menu. Then select **Break if X, N times...** from the List Selection dialog box, and click **OK**.

- Using the Break/Tracepoints pane, either:

  — select **Set BreakIf...** from the **Pane** menu

  — right-click on the pane background, and select **Set BreakIf...** from the context menu.

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

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

Simple Break if X, N times dialog box interface components

The Simple Break if X, N times dialog box (see Figure 4-11 on page 4-35) contains the following fields:

Enter Address Expression to Break on

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

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

Specify the location where the breakpoint is to be set. This can be:

- a specific line number in the source code, with or without a module name prefix (see *Qualifying breakpoint line number references with module names* on page 4-7)
- a specific address or address range (see *Specifying address ranges* on page 4-7)
- a function entry (see *Specifying the entry point to a function* on page 4-8).

The breakpoint unit triggers if the PC equals the corresponding address, or falls within the specified address range. For example, `Proc_4`.

Alternatively, you can click the drop-down arrow to the right of these fields to choose the location from a list browser (see *List browser dialog boxes* on page 8-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 8-32), or select from a list of previously-used expressions.

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

Note

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

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

The Simple Break if X, when Y is True dialog box, shown in Figure 4-12, enables you to set a conditional breakpoint that specifies an expression (given in C format). The result must be True when execution arrives at the specified address for the breakpoint to be triggered. This dialog box provides a quick way of assigning the **When Expression True** qualifier to a breakpoint (see *Specifying Qualifiers* on page 4-52).

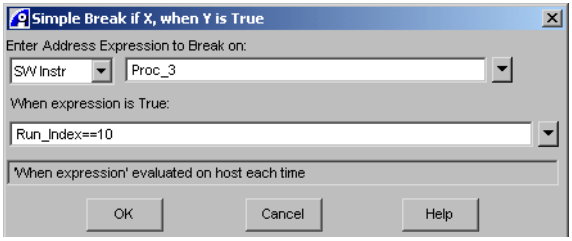

Figure 4-12 Simple Break if X, when Y is True dialog box

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

To display the Simple Break if X, when Y is True dialog box (see Figure 4-12) use one of the following methods:

- Select the following option from the Code window main menu:

  Debug → Breakpoints → Conditional → Break if X, when Y is True...

- Right-click on the margin, an instruction, or line of source code in the File Editor pane, and select **Set BreakIf...** from the context menu. Then select **Break if X, when Y is True...** from the List Selection dialog box, and click **OK**.

- Using the Break/Tracepoints pane, either:

  — select **Set BreakIf...** from the **Pane** menu

  — right-click on the pane background, and select **Set BreakIf...** from the context menu.

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

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

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

The Simple Break if X, when Y is True dialog box (see Figure 4-12 on page 4-37) contains the following fields:

Enter Address Expression to Break on

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

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

Specify the location where the breakpoint is to be set. This can be:

- a specific line number in the source code, with or without a module name prefix (see *Qualifying breakpoint line number references with module names* on page 4-7)
- a specific address or address range (see *Specifying address ranges* on page 4-7)
- a function entry (see *Specifying the entry point to a function* on page 4-8).

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

When expression is True

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

You can enter values directly in these fields, or you can click the drop-down arrow to the right of the fields to choose from a list browser (see *List browser dialog boxes* on page 8-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 8-32), or select from a list of previously-used expressions.

4.7.4 Setting conditional hardware breakpoints

Although hardware breakpoints can have conditions that are controlled in hardware, they are not conditional breakpoints in RealView Debugger. RealView Debugger recognizes a breakpoint as conditional only when a condition qualifier is assigned to the breakpoint. Condition qualifiers are software conditions that are controlled by RealView Debugger.

If you have created a hardware breakpoint using one of the methods described in *Setting hardware breakpoints explicitly* on page 4-24, then you must edit the breakpoint to assign one or more condition qualifiers to it. To do this, right-click on the breakpoint entry in the Break/Tracepoints pane, and select **Edit Break/Tracepoint...** from the context menu. The Set Address/Data Breakpoint dialog box is displayed, where you can assign condition qualifiers to the breakpoint.

For details on using the Set Address/Data Breakpoint dialog box, see *Using the Set Address/Data Breakpoint dialog box* on page 4-41.

4.7.5 Setting conditional breakpoints in other ways

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

- Use drag-and-drop to create a breakpoint in the Break/Tracepoints pane based on a program item, for example highlight a source code function in the File Editor pane and then drag it (using your mouse) and drop it into the Break/Tracepoints pane (see *Using the Break/Tracepoints pane* on page 4-63 for details).
- Set a breakpoint on a memory location in the Memory pane. The type of breakpoint offered depends on the type of memory at the chosen location, for example RAM, ROM, or Flash. See *Operating on memory contents* on page 6-25 for more details.
- Set a hardware breakpoint at an address of a symbol using the **Set Break At...** option on the Stack pane context menu, and assign a qualifier to the breakpoint. This displays the Set Address/Data Breakpoint dialog box (see *Using the Set Address/Data Breakpoint dialog box* on page 4-41).

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

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

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

  See *Operating on stack contents* on page 6-33 for more details on using the Stack pane.
- Right-click on a value shown in the Watch pane, or on a variable shown in the Call Stack pane, then select **Break at...** from the context menu. This displays the Set Address/Data Breakpoint dialog box (see *Using the Set Address/Data Breakpoint dialog box* on page 4-41).

See *Managing watches* on page 6-42 for more details on using the Watch pane.

See *Using the call stack* on page 6-36 for more details on using the Call Stack pane.

- You can use the Data Navigator pane to set a conditional breakpoint on a chosen function, defined as a location in the image. See *Using the Data Navigator pane* on page 8-5 for more details.

4.8 Using the Set Address/Data Breakpoint dialog box

The Set Address/Data Breakpoint dialog box provides comprehensive facilities to enable you to specify new breakpoints of any type in full (see *Breakpoint types* on page 4-2). You can also use it to edit existing breakpoints.

This section describes how to use the Set Address/Data Breakpoint dialog box, and includes:

- *Displaying the Set Address/Data Breakpoint dialog box*
- *Editing an existing breakpoint* on page 4-42
- *Set Address/Data Breakpoint dialog box interface components* on page 4-42
- *Setting software breakpoints* on page 4-45
- *Setting hardware breakpoints* on page 4-46
- *HW Support settings for RVISS* on page 4-47
- *HW Support settings for target hardware* on page 4-48
- *HW Support settings for a DSP-based debug target* on page 4-49
- *Using ranges and masks with hardware breakpoints* on page 4-50
- *Specifying Qualifiers* on page 4-52
- *Specifying Actions* on page 4-53
- *Managing the qualifier and action lists* on page 4-55.

4.8.1 Displaying the Set Address/Data Breakpoint dialog box

To display the Set Address/Data Breakpoint dialog box, shown in Figure 4-13 on page 4-42, use one of the following methods:

- Select **Debug → Breakpoints → Set/Edit Breakpoint...** from the Code window main menu.
- Using the Break/Tracepoints pane, either:
  - select **Set/Edit Breakpoint...** from the **Pane** menu
  - right-click on the pane background, and select **Set/Edit Breakpoint...** from the context menu.
- Right-click on the margin, an instruction, or line of source code in the File Editor pane, and select **Set Break...** from the context menu.
- Right-click on an memory location in the memory pane, and select **Set Break At...** from the context menu.
- Right-click on a value shown in the Watch pane, or on a variable shown in the Call Stack pane, then select **Break at...** from the context menu.

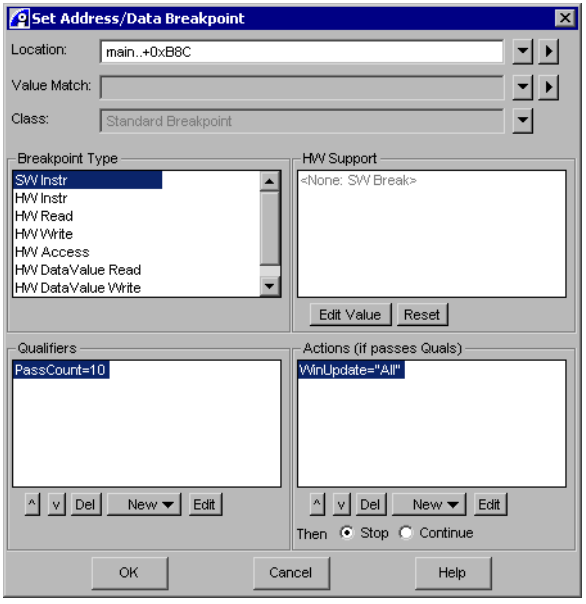

Figure 4-13 Set Address/Data Breakpoint dialog box

4.8.2 Editing an existing breakpoint

If you want to edit an existing breakpoint, right-click on the breakpoint entry in the Break/Tracepoints pane, and select **Edit Break/Tracepoint...** from the context menu. The Set Address/Data Breakpoint dialog box is displayed, and is populated with the current breakpoint attributes.

This is useful, for example, if you have previously set a default breakpoint, and you want to assign a condition qualifier or action to that breakpoint.

4.8.3 Set Address/Data Breakpoint dialog box interface components

The main interface components of the Set Address/Data Breakpoint dialog box (see Figure 4-13) are:

Location Specifies the memory location where the new breakpoint is set. This can be:

- a specific line number in the source code, with or without a module name prefix (see *Qualifying breakpoint line number references with module names* on page 4-7)
- a specific address or address range (see *Specifying address ranges* on page 4-7)

- a function entry (see *Specifying the entry point to a function* on page 4-8).

You can enter the location directly in the field, or click the drop-down arrow to the right of this field to choose a location from a list browser (see *List browser dialog boxes* on page 8-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 8-32), or select from a list of previously-used expressions. The options shown here depend on your debug target and connection.

Where supported by your target hardware, use the options from the right-arrow menu to qualify the location (see *Using ranges and masks with hardware breakpoints* on page 4-50 for details).

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

Value Match

Enter the data value that triggers the breakpoint. Alternatively, click the drop-down arrow to the right of this field to choose from a list browser (see *List browser dialog boxes* on page 8-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 8-32), or select from a list of previously-used expressions.

If you use this with data breakpoints, this compares the data value that is read or written.

Where supported by your target hardware, use the options from the right-arrow menu to qualify the value match (see *Using ranges and masks with hardware breakpoints* on page 4-50 for details).

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

Class A read-only field to show the type of breakpoint you set.

By default, RealView Debugger sets a `Standard Breakpoint` but the contents of this field change depending on your debugging environment and target configuration. For example, the Class field might show `Thread Breakpoint`.

See the chapters that describe tracing and RTOS support in *RealView Debugger v1.8 Extensions User Guide* for more details on breakpoint classes, and how to work with different types of breakpoint or tracepoint.

Breakpoint Type

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

If you select a Breakpoint Type, the contents of other groups in this dialog box might change. For example, if your current target supports hardware breakpoints, and you select a hardware breakpoint type, the HW Support group is populated with one or more options.

For details, see:

- *Setting software breakpoints* on page 4-45
- *Setting hardware breakpoints* on page 4-46.

HW Support

If your current target supports breakpoint tests in hardware, and you select a hardware Breakpoint Type, this group is populated with a list of currently available tests.

For details on using these controls, see:

- *HW Support settings for RVISS* on page 4-47
- *HW Support settings for target hardware* on page 4-48.
- *HW Support settings for a DSP-based debug target* on page 4-49.

Qualifiers When setting a conditional breakpoint, you specify the condition that must be satisfied to trigger the breakpoint. Qualifiers are the tests that RealView Debugger carries out to trigger the breakpoint. If you specify more than one qualifier, then all those specified must be met before the breakpoint triggers. Click **New** to display the **New Qualifiers** menu, where you can define the test criteria. This menu contains the following options:

- **SW Pass Count...**
- **When Expression True...**
- **When Expression False...**
- **User Macro...**
- **C++ Object...**
- **Favorites...**

Any qualifiers you have set previously are listed below the **Favorites...** option.

See *Specifying Qualifiers* on page 4-52 for details on using these controls.

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

Click **New** to display the **New Actions** menu. This menu contains the following options:

- **Update Window...**
- **Update All Windows**
- **Update Sample...**
- **Breakpoint Timer**
- **Message Output...**
- **Favorites...**

Any actions you have set previously are listed below the **Favorites...** option.

See *Specifying Actions* on page 4-53 for details on using these controls.

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

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

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

Note

Depending on the Breakpoint Type you select, the Location or Value Match fields might be unavailable. In this case, the field is grayed out. Also, in this example, the Class field is read-only and the drop-down arrow is unavailable.

4.8.4 Setting software breakpoints

To set a software breakpoint, select **SW Instr** as the Breakpoint Type. You can add qualifiers and actions as required. For more details, see:

- *Specifying Qualifiers* on page 4-52
- *Specifying Actions* on page 4-53.

4.8.5 Setting hardware breakpoints

Select a hardware Breakpoint Type to display entries specific to the hardware support for breakpoints available on the current target:

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

HW Read Sets or modifies a read breakpoint at the specified memory location or address range (see *Specifying address ranges* on page 4-7). The breakpoint is triggered if the application reads from any part of the specified memory range. Where supported by your debug target, you can also add data tests.

HW Write Sets or modifies a write breakpoint at the specified memory location or address range (see *Specifying address ranges* on page 4-7). The breakpoint is triggered if the application writes to any part of the specified memory range.

HW Access Sets or modifies an access breakpoint at the specified memory location or address range (see *Specifying address ranges* on page 4-7). The breakpoint is triggered when a memory address is accessed. This type of breakpoint enables you to perform hardware tests or to compare data values.

HW Data Value Read

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

HW Data Value Write

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

HW Data Value Access

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

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

- *HW Support settings for RVISS* on page 4-47
- *HW Support settings for target hardware* on page 4-48

- *HW Support settings for a DSP-based debug target* on page 4-49.

To set up the hardware breakpoint you must enter:

Location: Specifies the memory location where the new breakpoint is set.

Where supported by your target hardware, use the options from the right-arrow menu to qualify the location (see *Using ranges and masks with hardware breakpoints* on page 4-50 for details).

Value Match:

Enter the data value that triggers the breakpoint.

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

Where supported by your target hardware, use the options from the right-arrow menu to qualify the value match (see *Using ranges and masks with hardware breakpoints* on page 4-50 for details).

Note

Depending on the Breakpoint Type you select, the Location or Value Match fields might be unavailable. In this case, the field is grayed out. Also, in this example, the Class field is read-only and the drop-down arrow is unavailable.

You can add qualifiers and actions as required. For more details, see:

- *Specifying Qualifiers* on page 4-52
- *Specifying Actions* on page 4-53.

4.8.6 HW Support settings for RVISS

If you are using the RealView Simulator Broker interface to RVISS, hardware breakpoints have a single HW Support group option, **HWPassCount**, shown in Figure 4-14. This is initially set to **Off**, but you can set this to an integer value.

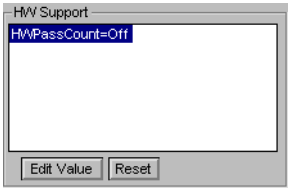

Figure 4-14 HW Support group using RVISS

The **HWPassCount** enables you to specify the number of times the test point is passed before the breakpoint triggers. To change the hardware pass count, click **Edit Value** to edit how the test is defined, shown in Figure 4-15.

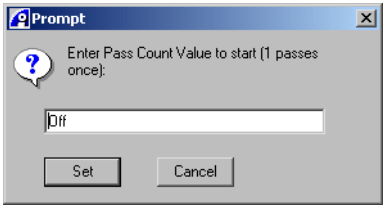

Figure 4-15 Changing the hardware pass count

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.

4.8.7 HW Support settings for target hardware

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

DataSize Supports testing **MAS** signals from the core. This enables you to test the size of *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 4-16 on page 4-49.

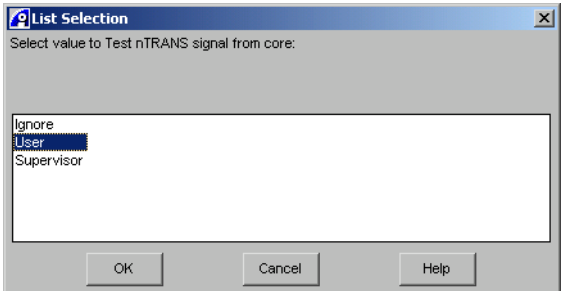

Figure 4-16 Configuring hardware tests

Click **OK** to confirm the new test and close the list selection box. The HW Support display list shows the new test.

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

4.8.8 HW Support settings for a DSP-based debug target

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

If you are working with a CEVA, Inc. DSP based debug target, the HW Support group offers a single option, shown in Figure 4-17. The `HWPassCount` enables you to specify the number of times the test point is passed before the breakpoint triggers.

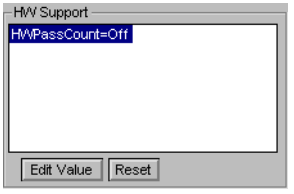

Figure 4-17 HW Support group using a DSP

To change the hardware pass count, click **Edit Value** to edit how the test is defined, shown in Figure 4-18 on page 4-50.

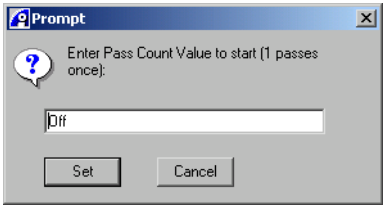

Figure 4-18 Changing the hardware pass count in a DSP

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 with hardware breakpoints* for details).

4.8.9 Using ranges and masks with hardware breakpoints

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

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

- **Address Range (HW Breaks only)**
- **Address Range by Length (HW Breaks only)**
- **Address Mask (HW Breaks only)**
- **NOT Address Compare (HW Breaks only)**
- **Autocomplete Range**.

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

- **Value Range (HW Breaks only)**
- **Value Range by Length (HW Breaks only)**
- **Value Mask (HW Breaks only)**
- **NOT Value Compare (HW Breaks only)**
- **Autocomplete Range**.

Note

These menu options are only available where supported by your debug target and if you have specified a hardware Breakpoint Type.

Choose from the available options to set up your hardware breakpoint:

Address/Value Range

Enter the start address, or data value, for the breakpoint then click this option to specify a range, for example the address range `0x800FF..0x80A00` (see *Specifying address ranges* on page 4-7). The separators `..` are automatically inserted for you.

Address/Value Range by Length

Enter the start address, or data value, for the breakpoint then click this option to specify a range by length, for example the address range `0x800FF..+0x1111` (see *Specifying address ranges* on page 4-7). The separators `..+` are automatically inserted for you.

Address/Value Mask

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

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

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

NOT Address/Value Compare

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

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

Autocomplete Range

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

Note

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

4.8.10 Specifying Qualifiers

To set a conditional breakpoint, you must assign a condition qualifier. To do this, click the **New** button in the Qualifiers group, shown in Figure 4-19, to display the **New Qualifiers** menu. You can use this to specify qualifiers when you first create a breakpoint, or to add qualifiers to edit an existing breakpoint.

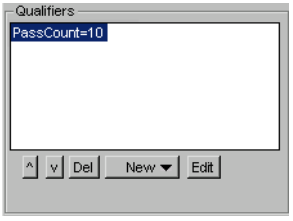

Figure 4-19 Assigning qualifiers to breakpoints

The order of the qualifiers, in the Qualifiers group display list, defines the order they are tested to trigger the breakpoint. If a condition is False, then subsequent conditions are not tested. See *Controlling the behavior of breakpoints* on page 4-59 for details on how the order of conditional qualifiers affects the behavior of a breakpoint.

The New Qualifiers menu

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

SW Pass Count...

Use this to specify the number of times execution must arrive at the specified address before execution stops.

The equivalent CLI command qualifier is `passcount:`*`count`*.

When Expression True...

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

The equivalent CLI command qualifier is `when:{`*`condition`*`}`.

When Expression False...

Use this to specify an expression that must evaluate to False to stop execution.

The equivalent CLI command qualifier is `when_not:{condition}`.

User Macro...

Enables you to specify a macro that runs when execution stops. This brings up a dialog box where you supply the macro name and any arguments required to run it. See *Attaching macros to breakpoints* on page 4-56 for an example. Also, see *Controlling the behavior of breakpoints* on page 4-59 for details on how the macro return value affects the behavior of a breakpoint.

The equivalent CLI command qualifier is `macro:{MacroCall(arg1,arg2)}`.

C++ Object...

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

The equivalent CLI command qualifier is `obj:(n)`.

Favorites... Select this option to display your personal Favorites List of breakpoint qualifiers (see *Favorites categories used by RealView Debugger features* on page 8-32). From here you can specify the required qualifier.

See *Setting breakpoints from your breakpoint Favorites List* on page 4-76 for details of creating breakpoint favorites and adding qualifiers to the list.

4.8.11 Specifying Actions

You can assign actions to your breakpoint. To do this, click the **New** button in the Actions group, shown in Figure 4-20, to display the **New Actions** menu. You can use this to specify breakpoint actions when you first create a breakpoint, or to add actions to edit an existing breakpoint.

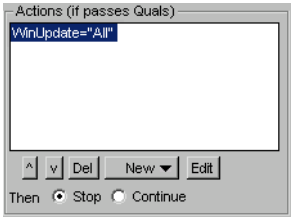

Figure 4-20 Assigning actions to breakpoints

The order of the actions, in the Actions group display list, defines the order they are carried out when the breakpoint triggers.

Actions do not execute until all condition qualifiers, if any, complete successfully. If no condition qualifiers are specified, the actions are always performed. See *Controlling the behavior of breakpoints* on page 4-59 for details on how actions are performed for a breakpoint when used with condition qualifiers.

The New Actions menu

The **New Actions** menu enables you to specify one or more actions to be performed when a breakpoint triggers, and any condition qualifiers, if any, complete successfully:

Update Window...

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

You can only use this selection box to specify one window at a time. If you want to update several windows, repeat the operation to set up a windows list. The list order specifies the update order.

The equivalent CLI command qualifier is `update:{"name"}`.

Update All Windows

Updates all desktop windows when the breakpoint triggers.

The equivalent CLI command qualifier is `update:{"all"}`.

Update Sample...

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

Note

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

Breakpoint Timer

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

The equivalent CLI command qualifier is `timed`.

Message Output...

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

By default this message appears in the Output pane but you can specify a the ID of a user defined window or an open file. For example, if you have a user defined window with an ID of `250`, then `$250$Stop at convert proc` sends the message `Stop at convert proc` to that window (see *Attaching macros to breakpoints* on page 4-56 for an example).

The equivalent CLI command qualifier is `message:{"$windowid | fileid$message"}`.

Favorites... Select this option to display your personal Favorites List of breakpoint actions (see *Favorites categories used by RealView Debugger features* on page 8-32). From here you can specify the required action.

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

Setting the continuation state

Use the **Stop** and **Continue** options to specify whether the program is to continue or stop executing when the breakpoint is triggered. The default state is to stop execution. Any specified action qualifiers are still performed, depending on the results of any condition qualifiers.

The **Continue** option is equivalent to the `continue` command qualifier of the breakpoint CLI commands. The **Stop** option does not have an equivalent command qualifier.

See *Controlling the behavior of breakpoints* on page 4-59 for details on how the continuation state affects the behavior of a breakpoint.

4.8.12 Managing the qualifier and action lists

To manage the Qualifiers and Actions display lists:

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

Qualifiers, actions, and the continuation state, are set up using the Set Address/Data Breakpoint dialog box, shown in Figure 4-13 on page 4-42.

4.9 Attaching macros to breakpoints

RealView Debugger includes a macro facility that enables you to create macros containing complex procedures. You can execute these macros directly through the RealView Debugger CLI, or you can attach a macro to a breakpoint so that it is executed when the breakpoint triggers. When you attach a macro to a breakpoint, the macro can return an integer value that determines whether program execution continues or stops. See *Controlling the behavior of breakpoints* on page 4-59 for details on how the macro return value affects the behavior of a breakpoint.

RealView Debugger recognizes several predefined macros containing commonly used functions. These macros can also be attached to breakpoints. However, if you are attaching a macro that you create yourself, for example the `tutorial()` macro created in *Using macros* on page 9-8, then this must be defined as a symbol in the debugger.

Macro syntax is defined in the *RealView Debugger v1.8 Command Line Reference Guide*.

Note

Execution-type commands are not valid within a breakpoint macro. See *Using macros with breakpoints* on page 9-6 for full details.

This section includes:

- *Defining a user-created macro as a symbol*
- *Attaching a macro to a breakpoint* on page 4-57.

4.9.1 Defining a user-created macro as a symbol

To define a user-created macro as a symbol, so that you can attach it to a breakpoint:

1. Make sure the macro is defined in an include file, for example `tutorial.inc`.
2. Select **Tools → Include Commands from File...** to display the Select File to Include Commands from dialog box.
3. Locate and highlight the macro file. That is, `tutorial.inc` in this example.
4. Click **Open** to load the macro file into the debugger. The debugger defines the macro as a symbol with the same name as the macro, for example `tutorial`.

4.9.2 Attaching a macro to a breakpoint

You can attach a predefined macro, or a macro you have created yourself, to a breakpoint. If you want to attach a macro you have created yourself, you must first load the macro definition into the debugger as described in *Defining a user-created macro as a symbol* on page 4-56.

To attach a macro to a breakpoint:

1. Display the Set Address/Data Breakpoint dialog box using one of the methods described in *Displaying the Set Address/Data Breakpoint dialog box* on page 4-41.
2. Enter the location of the breakpoint, for example `0x8704`.
3. Click the **New** button in the Qualifiers group to display the **New Qualifiers** menu.
4. Select the option **User Macro...** to display the data entry prompt, shown in Figure 4-21.

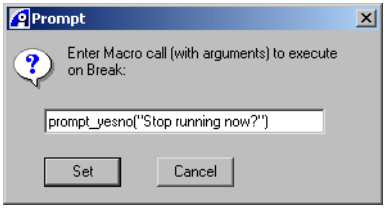

Figure 4-21 Breakpoint macro entry prompt

5. Enter the macro name and arguments, if any.

   The predefined macro shown in Figure 4-21 displays a message box to halt execution if the **Yes** key is pressed.
6. Click **Set** to confirm your entry.
7. The Set Address/Data Breakpoint dialog box now contains the macro in the Qualifiers group for the conditional breakpoint, shown in Figure 4-22.

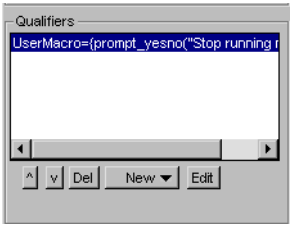

Figure 4-22 Set breakpoint with macro attached

8. Click **OK** to confirm the breakpoint settings and close the dialog box.
9. Click **Run** to execute the program and trigger the breakpoint.

4.10 Controlling the behavior of breakpoints

You can control the behavior of a breakpoint by assigning one or more condition qualifiers and actions to that breakpoint. The order that you add the condition qualifiers and actions determines the order they are processed by RealView Debugger. However, actions are performed only when the result of all specified condition qualifiers is True, even if you specify any actions before a condition qualifier.

This section includes:

- *Demonstrating breakpoint behavior with the Dhrystone example project*
- *Using a macro as an argument to a break command* on page 4-61.

4.10.1 Demonstrating breakpoint behavior with the Dhrystone example project

To demonstrate the behavior of a breakpoint using the Dhrystone example project:

1. Load the `dhrystone.prj` project file, and rebuild the image if necessary.
2. Connect to a debug target.
3. Load the `dhystone.axf` image.
4. Set a software breakpoint in `dhry_1.c`,at column 2 of line 153. Although you can use the RealView Debugger GUI to set breakpoints, it is easier to understand the interaction of qualifiers and actions using CLI commands (see the chapter that describes the CLI commands in the *RealView Debugger v1.8 Command Line Reference Guide*).

   Use the following command:

   `breakinstruction,`*`qualifiers`* `(I A)\DHRY_1\#153:2`

   For *`qualifiers`* use the following (see the command qualifiers in Table 4-3 on page 4-60, for each command qualifier variation):

   - A macro to view a symbol, for example:

     ```
     define /R int viewSymbol()
     {
       $printsymbol DHRY_1\clock_t$;
       return (0); //   0 - stop execution at the breakpoint
                   // >=1 - continue execution
     }
     .
     ```

     See Chapter 9 *Working with Macros* for more details on defining and using macros. Also, see *Using a macro as an argument to a break command* on page 4-61.
   - A pass count specifying five passes before the breakpoint is to be triggered.
   - An action qualifier that prints the message `Actions performed`.

5. Run the image, enter `10` when prompted for the number of runs.
6. If you are using the RealView Debugger GUI, click the **Cmd** tab in the Output pane to view the results.
7. Repeat steps 3 to 6 for each command qualifier variation, and change the macro return value as indicated.

Table 4-3 summarizes the breakpoint behavior for various combinations of the command qualifiers.

Table 4-3 Breakpoint behavior with multiple condition qualifiers and actions

Command qualifiers	Macro return value	Execute macro	Print message	Record the breakpoint details
`,message:{"Actions performed"},passcount:5`			After five passes	After five passes
`,passcount:5,message:{"Actions performed"}`			After five passes	After five passes
`,message:{"Actions performed"}` `,passcount:5,continue`			After five passes	Never
`,passcount:5,message:{"Actions performed"}` `,continue`			After five passes	Never
`,macro:{viewSymbol()},passcount:5` `,message:{"Actions performed"}`	`0` (stop)	After every pass	After five passes	After five passes
`,passcount:5,macro:{viewSymbol()}` `,message:{"Actions performed"}`	`0` (stop)	After five passes	After five passes	After five passes
`,macro:{viewSymbol()},passcount:5` `,message:{"Actions performed"},continue`	`0` (stop)	After every pass	After five passes	Never
`,passcount:5,macro:{viewSymbol()}` `,message:{"Actions performed"},continue`	`0` (stop)	After five passes	After five passes	Never
`,macro:{viewSymbol()},passcount:5` `,message:{"Actions performed"}`	nonzero (continue)	After every pass	Never	Never
`,passcount:5,macro:{viewSymbol()}` `,message:{"Actions performed"}`	nonzero (continue)	After five passes	Never	Never
`,macro:{viewSymbol()},passcount:5` `,message:{"Actions performed"},continue`	nonzero (continue)	After every pass	Never	Never
`,passcount:5,macro:{viewSymbol()}` `,message:{"Actions performed"},continue`	nonzero (continue)	After five passes	Never	Never

Messages similar to the following are displayed when a breakpoint is recorded:


```
Stopped at 0x000084C4 due to SW Instruction Breakpoint
Stopped at 0x000084C4: DHRY_1\main Line 153
```

From Table 4-3 on page 4-60 you can see that:

- The details of the breakpoint are never recorded when the breakpoint continues. That is when the return value of a macro is nonzero, or if a `continue` action is assigned.
- When the return value of a macro is nonzero, any actions are never performed, so the `Actions performed` message is never printed. In addition, any condition qualifiers that are specified after the macro that returns nonzero are never tested. Therefore, when using macros with breakpoints, make sure that you position the macros appropriately, especially if any return a nonzero value.
- The frequency that a macro is executed depends on whether it appears before or after other qualifiers.

Note

The `continue` action qualifier and a macro nonzero value result in different behavior. Although both inhibit the display of the breakpoint details, actions are never performed when a macro returns a nonzero value.

4.10.2 Using a macro as an argument to a break command

You can optionally specify a macro as an argument at the end of a break command. Any macro that is specified in this way is treated as being specified last in the command qualifier list. For example:

```
breakinstruction,passcount:5,message:{"Actions performed"} (I A)\DHRY_1\#153:2
;viewSymbol()
```

This command gives the same behavior where the command qualifiers are in the following order (see Table 4-3 on page 4-60):

```
,passcount:5,macro:{viewSymbol()},message:{"Actions performed"}
```

Because a macro argument is treated last in the command qualifier list, if you also specify a macro as a command qualifier, then execution of the macro argument depends on the result returned by the macro qualifier. For example, you might define a second macro, such as:

```
define /R int viewValue()
{
  $printf "Int_1_Loc: %d", Int_1_Loc$;
  return (0); //   0 - stop execution at the breakpoint
```

```
                // >=1 - continue execution
}
.
```

Specify this macro as a command argument, and the `viewSymbol()` macro as a command qualifier, for example:

```
breakinstruction,passcount:5,macro:{viewSymbol()},message:{"Actions performed"}
(I A)\DHRY_1\#153:2 ;viewValue()
```

The `viewValue()` macro runs only if `viewSymbol()` returns a value of zero. In addition, if `viewValue()` returns a nonzero value, then the `Actions performed` message is not displayed, and the breakpoint details are not recorded.

4.11 Using the Break/Tracepoints pane

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

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

Note

The Break/Tracepoints pane also enables you to manage tracepoints during Trace and Analysis operations, and to work with thread breakpoints when running in RSD mode. For full details see *RealView Debugger v1.8 Extensions User Guide*.

This section describes:

4.11.1 Displaying the Break/Tracepoints pane

Select **View → Break/Tracepoints** from the Code window main menu to display the Break/Tracepoints pane, shown in Figure 4-23.

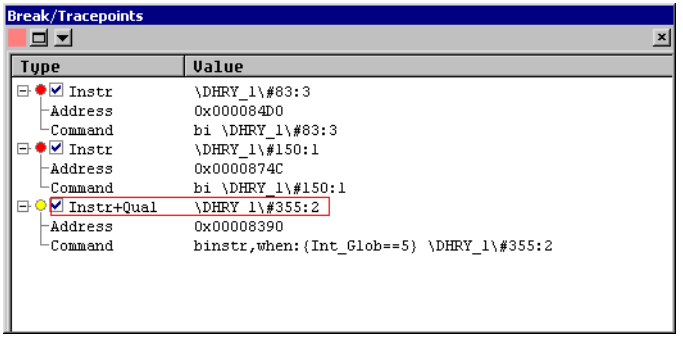

Figure 4-23 Break/Tracepoints pane

If any breakpoints are set already, these are displayed in the new pane. To expand the tree and see the breakpoint details either:

- Click on the plus sign associated with a particular breakpoint entry.
- Double-click anywhere along the breakpoint entry (not on the check box).

The Break/Tracepoints pane also displays any other breakpoints you set, for example tracepoints or thread breakpoints.

If no breakpoints are set then the Break/Tracepoints pane is empty. As you create and set new breakpoints (or tracepoints), the pane is automatically updated to display each new breakpoint.

4.11.2 Viewing breakpoints in the Break/Tracepoints pane

The Break/Tracepoints pane shows entries in a tree view giving the `Type` and `Value` of each breakpoint you set, shown in the example in Figure 4-24.

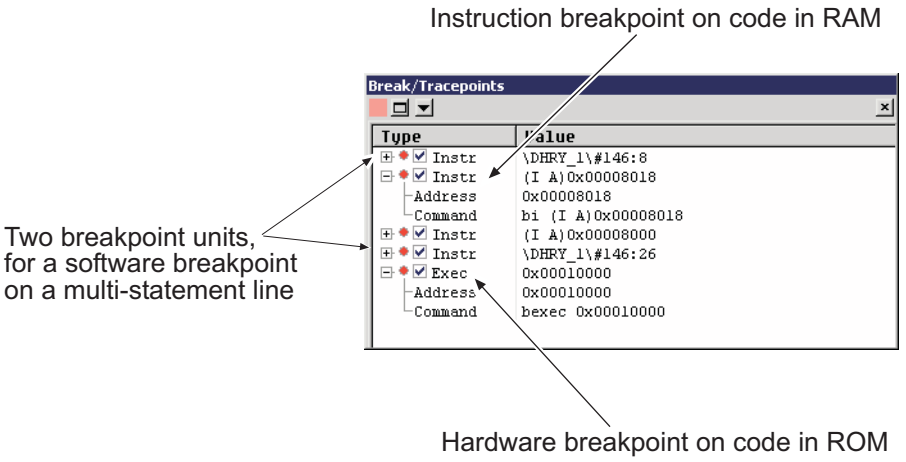

Figure 4-24 Breakpoints in the Break/Tracepoints pane

Each breakpoint is identified by a:

- colored icon to show the breakpoint type (see *Viewing breakpoints in your code view* on page 4-10 for details)
- check box to show if the breakpoint is enabled or disabled. You can click on the check box to change the state of the chosen breakpoint or breakpoint unit.

In Figure 4-24, an area of memory on the debug target has been designated ROM and so a `BREAKEXECUTION` command is used to set a hardware breakpoint. When the hardware breakpoint limit is reached for this debug target no more breakpoints can be set on code in this area, and a warning message is displayed.

Note

For details on the `BREAKINSTRUCTION` and `BREAKEXECUTION` commands see *RealView Debugger v1.8 Command Line Reference Guide*.

In the example in Figure 4-25 on page 4-66, different types of hardware breakpoints have been set, and disabled, during the session.

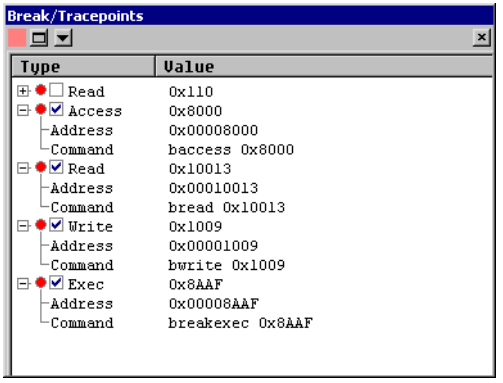

Figure 4-25 Hardware breakpoints in the Break/Tracepoints pane

4.11.3 Using the Break/Tracepoints Entry context menu

If you have a breakpoint set, right-click anywhere along the breakpoint entry to display the **Break/Tracepoints Entry** context menu.

The **Break/Tracepoints Entry** context menu includes the following options to edit and manage your breakpoints:

Edit Break/Tracepoint...

For breakpoints, displays the Set Address/Data Breakpoint dialog box, where you can edit the breakpoint details (see *Editing a breakpoint* on page 4-69).

For tracepoints, displays the Set/Edit tracepoint dialog box, where you can edit the tracepoint details (see the chapter that describes tracing with RealView Debugger in the *RealView Debugger v1.8 Extensions User Guide*).

Copy Break/Tracepoint...

For breakpoints, displays the Set Address/Data Breakpoint dialog box with details of the original breakpoint (see *Copying an existing breakpoint* on page 4-69).

For tracepoints, displays the Set/Edit tracepoint dialog box with details of the original tracepoint (see the chapter that describes tracing with RealView Debugger in the *RealView Debugger v1.8 Extensions User Guide*).

Clear Removes the breakpoint or tracepoint selected in the Break/Tracepoints pane.

Disable This option changes the status of a selected breakpoint or tracepoint.

Select **Disable** to disable a breakpoint or tracepoint you have set. This changes to **Enable** so that you can enable a breakpoint or tracepoint that has been disabled.

Select, or unselect, the check box in the Break/Tracepoints pane to change the status in the same way.

Reset PassCounters/Then-Enables

Resets the counters that are used to register the number of times that the program passes a breakpoint before it is triggered. If the breakpoint is enabled, then it stops being triggered until the count value is reached again.

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

Break/Tracepoint Favorites...

Displays the Favorites Chooser/Editor where you can edit or delete entries, or select from your favorites to set a breakpoint or tracepoint (see *Favorites categories used by RealView Debugger features* on page 8-32).

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

- an open blue pointer marks the location in source-level view
- a filled blue pointer marks the location in disassembly-level view if there is no source.

4.11.4 Using the Break/Tracepoints context menu

Right-click anywhere on the background of the Break/Tracepoints pane to display the **Break/Tracepoints** context menu. This context menu includes the following options to edit and manage your breakpoints:

Set Break from Function/Label list

Displays the Function Breakpoint/Profile Selector dialog box (see *Setting breakpoints from saved lists* on page 4-74).

Set/Edit Breakpoint...

Displays the Set Address/Data Breakpoint dialog box (see *Using the Set Address/Data Breakpoint dialog box* on page 4-41).

Set BreakIf...

Displays a List Selection dialog box containing a list of the breakpoints you can set. Each option displays a dialog box to set the breakpoint, as described in:

- *Setting unconditional breakpoints explicitly* on page 4-21
- *Setting hardware breakpoints explicitly* on page 4-24
- *Setting conditional breakpoints* on page 4-34.

Set Tracepoint...

Displays the Set/Edit Tracepoint dialog box (see the chapter that describes tracing with RealView Debugger in the *RealView Debugger v1.8 Extensions User Guide*).

Processor Exceptions...

Displays a List Selection dialog box containing a list of processor exceptions that you can enable or disable.

Break/Tracepoint History...

Displays a List Select dialog containing a list of breakpoints or tracepoints that you have previously set (see *Setting breakpoints from saved lists* on page 4-74).

Break/Tracepoint Favorites...

Displays a List Select dialog containing a list of breakpoint or tracepoint favorites that you have previously defined (see *Favorites categories used by RealView Debugger features* on page 8-32 and *Setting breakpoints from saved lists* on page 4-74).

Show Break Capabilities of HW...

Displays information about the hardware breakpoint support on the current debug target.

Clear All Break/Tracepoints

Clears all breakpoints and tracepoints defined for the current debug target.

4.11.5 Working in the Break/Tracepoints pane

If you have a breakpoint set, you can use the Break/Tracepoints pane to edit or copy a breakpoint, as described in:

- *Editing a breakpoint* on page 4-69
- *Copying an existing breakpoint* on page 4-69.

You can also use the Break/Tracepoints pane to edit or copy other breakpoints you set, for example tracepoints or thread breakpoints.

Editing a breakpoint

To edit a breakpoint:

1. Right-click on a breakpoint in the list to display the **Break/Tracepoints Entry** menu (see *Using the Break/Tracepoints Entry context menu* on page 4-66).
2. Select **Edit Break/Tracepoint...** from the context menu to display the Set Address/Data Breakpoint dialog box where you can edit the breakpoint or breakpoint unit (see *Using the Set Address/Data Breakpoint dialog box* on page 4-41).

When you edit a breakpoint, the breakpoint command includes the `modify` command qualifier (see the *RealView Debugger v1.8 Command Line Reference Guide*for more details).

Copying an existing breakpoint

To copy an existing breakpoint and create a new breakpoint:

1. Right-click on a breakpoint in the list to display the **Break/Tracepoints Entry** menu (see *Using the Break/Tracepoints Entry context menu* on page 4-66).
2. Select **Copy Break/Tracepoint...** from the context menu to display the Set Address/Data Breakpoint dialog box, populated with the relevant information about the chosen breakpoint (see *Using the Set Address/Data Breakpoint dialog box* on page 4-41).
3. Change the breakpoint location, to create the new breakpoint.

4.11.6 Using the Pane menu

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

Table 4-4 Breakpoint menu mappings

Break/Tracepoints pane, Pane menu	Code window, Debug menu
Set Break from Function/Label list	**Breakpoints → Set Break/Tracepoint from List → Set from Function/Label list...**
Set/Edit Breakpoint...	**Breakpoints → Set/Edit Breakpoint...**
Set BreakIf...	**Breakpoints → Conditional** and **Breakpoints → Hardware**
Set Tracepoint... [a]	**Tracepoints → Set Tracepoint...**
Processor Exceptions...	**Processor Exceptions...**
Break/Tracepoint History...	**Breakpoints → Set Break/Tracepoint from List → Break/Tracepoint History...**
Break/Tracepoint Favorites...	**Breakpoints → Set Break/Tracepoint from List → Break/Tracepoint Favorites...**
Show Break Capabilities of HW...	**Breakpoints → Hardware → Show Break Capabilities of HW...**
Clear All Break/Tracepoints	**Breakpoints → Clear All Break/Tracepoints**

a. See the chapter that describes tracing with RealView Debugger in the *RealView Debugger v1.8 Extensions User Guide* for details on setting tracepoints.

If you select the option **Set BreakIf...** from the **Pane** menu, a breakpoint type selection box is displayed containing all the breakpoints supported by your debug target. Each entry displays a dialog box to set the breakpoint, as described in:

- *Setting unconditional breakpoints explicitly* on page 4-21
- *Setting hardware breakpoints explicitly* on page 4-24
- *Setting conditional breakpoints* on page 4-34.

You can also access some of the options from the **Pane** menu by right-clicking on a blank area inside the Break/Tracepoints pane. This enables you to create new breakpoints, select from your personal favorites or history list, or to see your hardware support.

Note

The Break/Tracepoints pane also lists any tracepoints you have set. For full details on setting and using tracepoints, see the chapter that describes tracing in *RealView Debugger v1.8 Extensions User Guide*.

4.12 Disabling and clearing breakpoints

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

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

When you clear a breakpoint it is removed from the breakpoint list. To remove breakpoints from your Favorites List, use the Favorites Chooser/Editor dialog box, see *Setting breakpoints from your breakpoint Favorites List* on page 4-76 for details of how to do this.

This section describes:

- *Disabling breakpoints*
- *Clearing breakpoints* on page 4-73
- *Clearing all breakpoints* on page 4-73.

4.12.1 Disabling breakpoints

You can enable and disable breakpoints from the Code window:

- In the **Src** tab:
  1. Right-click in the left margin, or on the line number, of a line marked with a red breakpoint icon, or on the icon itself.
  2. Select **Disable Break** from the context menu.
- In the Break/Tracepoints pane, select the check box to unselect it so disabling the breakpoint.
- Right-click on the breakpoint in the Break/Tracepoints pane to display the **Break/Tracepoints** menu. Select **Disable** from the menu.
- Position the cursor at the breakpoint that you want to disable. Select **Debug** → **Breakpoints** → **Enable/Disable Break at Cursor** from the main menu.

You can use these methods to:

- enable a disabled breakpoint, marked by a white icon
- enable, or disable, a conditional breakpoint, marked by a yellow icon
- enable, or disable, tracepoints and thread breakpoints.

4.12.2 Clearing breakpoints

You can clear breakpoints from the Code window in different ways:

- In the **Src** tab, double-click on the line number of a line marked with a red breakpoint icon or on the icon itself.
- Right-click on the breakpoint in the Break/Tracepoints pane to display the **Break/Tracepoints** menu. Select **Clear** from the menu.
- Position the cursor at the breakpoint that you want to disable. Select **Debug** → **Breakpoints** → **Toggle Break at Cursor** from the main menu.

You can use these methods to clear any type of breakpoint, for example a conditional breakpoint (marked by a yellow icon), or a tracepoint, or a thread breakpoint.

Note

If supported by your debug target, a `RESET` command can be used to carry out a processor reset. If you are using RVISS, a `RESET` command does not clear breakpoints or tracepoints.

4.12.3 Clearing all breakpoints

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

- Select **Debug** → **Breakpoints** → **Clear All Break/Tracepoints** from the main menu.
- Click on the **Pane** menu in the Break/Tracepoints pane and select **Clear All Break/Tracepoints**.

4.13 Setting breakpoints from saved lists

This section describes how to set breakpoints from various lists held by RealView Debugger. If you use specific breakpoint definitions on a regular basis, then you can also define your own breakpoint lists. This section includes:

- *Setting breakpoints from the Function/Label list*
- *Setting breakpoints from the breakpoint history lists* on page 4-75
- *Setting breakpoints from your breakpoint Favorites List* on page 4-76.

Note

The lists described in this section can also be used for tracepoints. See the chapter that describes tracing with RealView Debugger in the *RealView Debugger v1.8 Extensions User Guide* for details on working with tracepoints.

4.13.1 Setting breakpoints from the Function/Label list

You can set a breakpoint on any number of the function names and labels in your image. The list of function names and labels in your image is available on the Function Breakpoint/Profile Selector dialog box. To display this dialog box, use one of the following methods:

- Select the following option from the Code window main menu:

  Debug → Breakpoints → Set Break/Tracepoint from List → Set from Function/Label list...

- Select the following option from the **Pane** menu of the Break/Tracepoints pane:

  Set Break from Function/Label list...

- Right-click on the background of the Break/Tracepoints pane, and select the following option from the context menu:

  Set Break from Function/Label list

Setting a breakpoint

To use the Function Breakpoint/Profile Selector dialog box to set a breakpoint on a chosen function:

1. Click on the check box for those functions where you want to set a breakpoint.
2. Click **Set** to set a breakpoint on the chosen functions.

Because the browser is used only to make a selection, there are no controls for debugging operations.

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

4.13.2 Setting breakpoints from the breakpoint history 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 include breakpoints you have set during the current debugging session, and breakpoints you have set from previous debugging sessions.

To see the current breakpoint history, use one of the following methods:

- Select the following option from the Code window main menu:

  Debug → Breakpoints → Set Break/Tracepoint from List... → Break/Tracepoint History...

- Select the following option from the **Pane** menu of the Break/Tracepoints pane:

  Break/Tracepoint History...

- Right-click on the background of the Break/Tracepoints pane, and select the following option from the context menu:

  Break/Tracepoint History...

Breakpoints are only added to the history list if they are set using breakpoint dialog boxes, for example the Set Address/Data Breakpoint or the Simple Break if X dialog box. If you set a breakpoint in another way, for example using a CLI command, this is not added to the list. To force this type of breakpoint to be added to the history list:

1. Set the required breakpoint using the appropriate CLI command, for example:

   ```
   bi \DHRY_1\#153:1
   ```

2. Select **View → Break/Tracepoints** from the Code window main menu to display the Break/Tracepoints pane, shown in Figure 4-23 on page 4-64.

3. Right-click anywhere along the required breakpoint entry (not on the check box) to display the **Break/Tracepoints Entry** context menu (see *Using the Break/Tracepoints Entry context menu* on page 4-66).

4. Select **Edit Break/Tracepoint...** to display the Set Address/Data Breakpoint dialog box.

5. Click **OK** to close the dialog box without changing the breakpoint details.

See *Setting breakpoints from your breakpoint Favorites List* for details of creating breakpoint favorites and adding existing breakpoints to your personal Favorites List.

Note

If you are using RealView Debugger on non-Windows platforms, the history file is only created if you create and save a favorite, for example a breakpoint or watchpoint. See Appendix B *RealView Debugger on Sun Solaris and Red Hat Linux* for details.

4.13.3 Setting breakpoints from your breakpoint Favorites List

When you first start to use RealView Debugger on Windows, your personal Favorites List is empty. You can create breakpoints and add them directly to this list or you can add breakpoints that you have been using in the current debugging session.

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

Note

You must connect to a target to enable RealView Debugger favorites.

You create and edit breakpoint favorites using the Favorites Chooser/Editor dialog box. This section describes how to use this dialog box, and contains:

- *Displaying the Favorites Chooser/Editor dialog box*
- *Creating a new breakpoint favorite* on page 4-77
- *Saving existing breakpoints as favorites* on page 4-77.

Displaying the Favorites Chooser/Editor dialog box

Use one of the following methods to display the Favorites Chooser/Editor dialog box:

- Select the following option from the Code window main menu:

  Debug → Breakpoints → Set Break/Tracepoint from List → Break/Tracepoint Favorites...

- Select the following option from the **Pane** menu of the Break/Tracepoints pane:

  Break/Tracepoint Favorites...

- Right-click on the background of the Break/Tracepoints pane, and select the following option from the context menu:

Break/Tracepoint Favorites...

For full details on how to use the Favorites Chooser/Editor dialog box, see *Using the Favorites Chooser/Editor dialog box* on page 8-29.

Creating a new breakpoint favorite

To create a new breakpoint and add it to your Favorites List:

1. Display the Favorites Chooser/Editor dialog box as described in *Displaying the Favorites Chooser/Editor dialog box* on page 4-76.
2. Click **New** to display the New/Edit Favorite dialog box.
3. Enter the CLI command to create the breakpoint and, optionally, a short text description to identify the breakpoint for future use.
4. Click **OK** to confirm the entries and close the New/Edit Favorite dialog box.

   The Favorites Chooser/Editor dialog box is displayed with the newly-created breakpoint shown in the display list.
5. Use the available controls to update the new breakpoint favorite (see *Favorites Chooser/Editor dialog box interface components* on page 8-30).

Saving existing breakpoints as favorites

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

To add existing breakpoints to your Favorites List:

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

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

5. Click **OK** to confirm the breakpoint details and close the dialog box.

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

The edited Favorites List is saved to your `exphist.sav` file, in your home directory, when you close RealView Debugger. For full details about the history file, see the chapter that describes how to end your session in *RealView Debugger v1.8 Essentials Guide*.

Note

If you are using RealView Debugger on non-Windows platforms, the history file is only created if you create and save a favorite, for example a breakpoint or watchpoint. See Appendix B *RealView Debugger on Sun Solaris and Red Hat Linux* for details.

Chapter 5
Memory Mapping

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

5.1 About memory mapping

Memory mapping is disabled by default when you first connect to your debug target, that is all memory is treated as RAM. If you are working with a suitable target you can enable memory mapping and then configure the memory using the **Map** tab in the Process Control pane. The memory map is usually configured in a BCD file that you associate with the connection (see *RealView Debugger v1.8 Target Configuration Guide* for more details).

This section includes:

- *Uses for memory mapping*
- *Memory map considerations* on page 5-3.

5.1.1 Uses for memory mapping

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

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

5.1.2 Memory map considerations

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

- The `top_of_memory` value must be higher than the sum of the program base address and program size. If set incorrectly, your program might crash because of stack corruption or because your program overwrites its own code.
- There is no requirement that the address specified by `top_of_memory` is at the true top of memory. A C or assembler program can use memory at higher addresses.
- If you are working with a scatterloaded application, you must define the location of stack and heap in your code.
- The default memory model for *RealView ARMulator® ISS* (RVISS) creates memory pages as required through the whole address space, and, therefore, the RVISS 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.
- By default, RealView Debugger tries to set a software breakpoint. However, where enabled, the memory map determines the type of breakpoint that you can set. For example, only hardware breakpoints can be set in Flash and ROM.
- The memory map is used to define how memory contents are color coded when displayed in the Memory pane.

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 → Memory Map tab** to display the Process Control pane and bring the **Map** tab to the front. The pane is displayed as a floating pane.

   Alternatively, if you have an instance of the Process Control pane that is docked, select the **Map** tab in that pane.
3. Right-click anywhere in the **Map** tab to display the **Map** context menu (see *Using the Map tab context menu* on page 5-10).
4. Click on the option **Toggle Memory Mapping** so that it is checked.

   This enables memory mapping ready to load your image.
5. To disable memory mapping, right-click anywhere in the **Map** tab to display the **Map** context menu.

   The context menu contains different options depending on the state of your debug target. For example, if you load an image the menu contains the option **Update Map based on Image**.
6. Click on the option **Toggle Memory Mapping** so that it is unchecked.

   This disables memory mapping ready to load your image.

   See *Using the Map tab context menu* on page 5-10 for details on the other options available from this menu.

Note

If you have assigned a board-chip definition file to a target connection, and that board-chip definition defines a memory map, then memory mapping is automatically enabled when you connect to that target. See the chapter that describes configuring custom targets in *RealView Debugger v1.8 Target Configuration Guide* for details of configuring your target this way.

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 disconnect from the target. See the chapter that describes configuring custom targets in *RealView Debugger v1.8 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 **Map** context menu (see *Using the Map tab context menu* on page 5-10).
2. Select **Add or Copy Map Entry...** to display the Add/Copy/Edit Memory Map dialog box, shown in Figure 5-1.

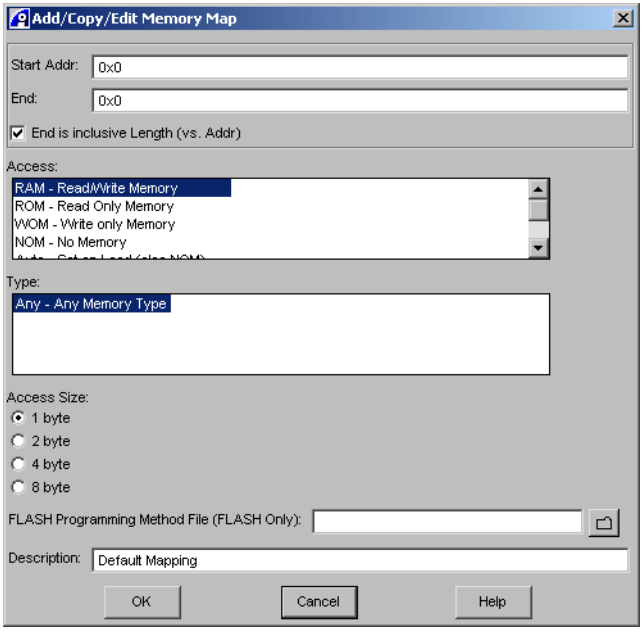

Figure 5-1 Add/Copy/Edit Memory Map dialog box

Note

In RealView Debugger, memory mapping is defined by a start address and a block size by default, not by an end address. If you want to specify the end address, you must unselect the **End is inclusive Length (vs Addr.)** check box.

3. Edit the dialog box to change the default memory mapping as follows:
   a. Enter `0x0` in the Start Addr field.
   b. Enter `0x8000` in the End field to specify the block size.
4. Enter `Area before image` in the Description field to describe this block.
5. Click **OK** to confirm your changes and the Process Control **Map** tab is updated.
6. Set up the second block of memory using these settings:
   a. Start address = `0x8000`
   b. Length = `0x8000`
   c. Description = `Middle`
7. Click **OK** to confirm your changes and the Process Control **Map** tab is updated.
8. Set up the third block of memory using these settings:
   a. Start address = `0x10000`
   b. Length = `0xFFFF0000`
   c. Description = `Area after image`
9. By default, memory access is set to byte-size (8 bits) for ARM processors. Do not change this.
10. Click **OK** to confirm your changes and update the Process Control **Map** tab, shown in Figure 5-2.

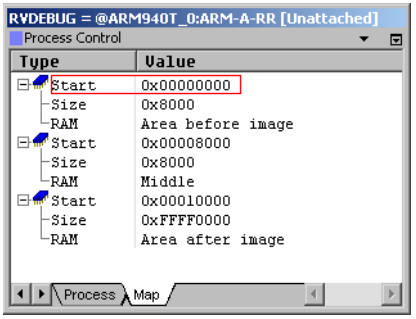

Figure 5-2 Memory mapped

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 5-10.

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 → Memory Map tab** to display the Process Control pane and bring the **Map** tab to the front. The pane is displayed as a floating pane.

   Alternatively, if you have an instance of the Process Control pane that is docked, select the **Map** tab in that pane.
5. Click on the plus signs to expand the entries, shown in Figure 5-3 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 target vehicle.

For example, if you are working with an ARM architecture-based target, the first entry in the memory map shows `Start` (see Figure 5-3 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 method file (`*.fme`) is present, writable.

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

Size Defines the size of the memory area.

Access size Defines the size of memory accesses.

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

Description A text description of the purpose of the memory supplied as part of the automatic mapping. You can also supply this information yourself (see *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 colored icons to make the display easier to read and to highlight the different memory definitions, shown in the example in Figure 5-3 on page 5-9.

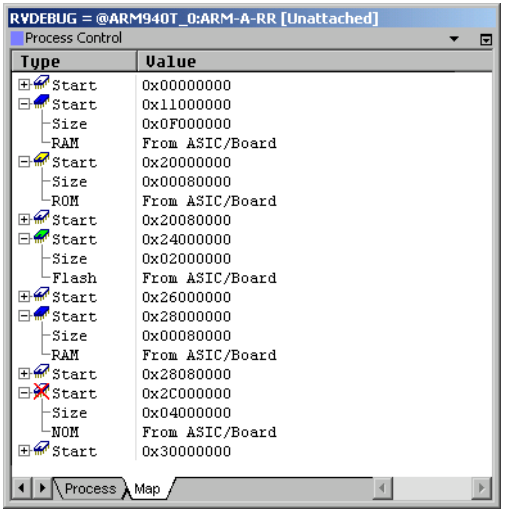

Figure 5-3 Colors in the memory map

In this example, memory has been mapped, using a Board/Chip definition file, to declare Flash. See the chapter that describes configuring custom targets in *RealView Debugger v1.8 Target Configuration Guide* for details of configuring your target this way.

Colored icons enable you to identify the memory access defined:

- white (open) indicates one of the following:
  - — no memory type is defined
  - — you have set the memory to `Auto`, that is, auto-define as RAM when image loads into it
  - — you have set the memory to `Prompt`, that is, prompt if image loads into it.
- blue indicates `RAM`
- yellow indicates `ROM`
- green indicates Flash memory known to RealView Debugger
- red indicates write-only memory (`WOM`)
- red cross indicates no accessible memory (`NOM`) is defined.

Also, see *Display colors* on page 5-8 for a description of the display colors in the Memory pane.

5.4.3 Using the Map tab context menu

The **Map** tab context menu enables you to add new mappings or to update existing ones. The options are:

Add or Copy Map Entry...

Displays the Add/Copy/Edit Memory Map dialog box, shown in Figure 5-1 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 5-1 on page 5-5, where you can edit a map entry.

Update Map based on Image

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

Update Map based on Processor

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

Save Map to Linker Command file...

Writes the current map state to a new or existing linker command file. This inserts or edits the `MEMORY` definitions in the linker command file, and enables the proper loading of an application based on actual memory settings. See *Generating linker command files for non-ARM targets* on page 5-14 for full details of how to generate this file for targets other than ARM targets.

Delete Map Entry

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

Reset Map (Delete All)

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

Toggle Memory Mapping

Enables or disables memory mapping. When checked, memory mapping is enabled.

Properties...

Displays information about the selected memory map entry.

5.5 Editing map entries

You can edit any memory map entry except for those that have default mapping. To edit a memory map entry using the **Map** tab:

1. Right-click on the map entry in the display list to display the context menu.
2. Select the option **Edit Map Entry...** to display the Add/Copy/Edit Memory Map dialog shown in Figure 5-1 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 `0xFFFFFFF0`.

   RealView Debugger automatically sets the size you specify. If the computed size does not fall on a page boundary an error dialog is displayed and you must resubmit the block size.

   Entering a value of `0x0` remaps all memory from the starting address.
5. Highlight the access type in the display list, for example `RAM`.
6. Enter the memory type to be allocated, for example `Any`.
7. Enter a description of the new memory map settings, for example `New test memory entry`.
8. Click **OK** to confirm your new settings and to update the **Map** tab.

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

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 **Map** context menu (see *Using the Map tab context menu* on page 5-10) 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 `top_of_memory`. Stack and base are then set to be immediately below `top_of_memory`.

RealView Debugger uses default settings for stack and base that are target-dependent. For both RVISS and ARM processors, the default setting for `top_of_memory` is `0x80000`.

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

```
Warning: No stack/heap or top of memory defined - setting top_of_memory to
0x80000.
```

To avoid this message, permanently set `top_of_memory` using the Connection Properties window. Configure this for your debug target so that it is used whenever you connect with RealView Debugger. See the chapter that describes configuring custom targets in *RealView Debugger v1.8 Target Configuration Guide* for details of how memory is configured in ARM architecture-based debug targets, and for an example of how to set up your memory map.

You can set `top_of_memory`, and other ARM runtime controls specific to ARM processors, as part of a project definition. However, the available options depend on your target processor and target vehicle.

You can also set `top_of_memory` 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 RVISS 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 when you build your program. This `MEMORY` directive information can then be used to position various sections of an application correctly. For details of how to set up such command options see the chapter that describes customizing projects in *RealView Debugger v1.8 Project Management User Guide*.

Note

Linker command files are not currently supported by ARM targets.

To generate or modify a linker command file:

1. Right-click on the start address at the top of the entries and select **Save Map to Linker Command File...** from the context menu.
2. Specify the location of the file in the Select Linker Command File to Create or Modify dialog box. Remember that:
   - If the file already exists, RealView Debugger looks for a `MEMORY` directive block created previously and, if found, replaces that block.
   - If the file already exists, but no `MEMORY` directive block exists, RealView Debugger locates the first `MEMORY` section and inserts the `MEMORY` directive block before it.
   - If the file already exists, RealView Debugger makes a backup copy before updating the contents.
   - If there is no existing file, RealView Debugger creates the specified file ready to accept the `MEMORY` directive block.

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

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

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

Example 5-1 on page 5-15 shows an example of a generated linker command file, in DSP format.

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 04 15:10:41 2003
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) in addition to external memory as defined by the loading of a program.

The following notes apply to this automatic file generation process:

- If RealView Debugger creates the linker command file a comments section is inserted in the file reporting that it was generated by RealView Debugger, shown in Example 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
Working with Debug Views

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

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.

The registers that are visible depend on your debug target, that is your processor and the debug interface. See your processor hardware documentation for details on processor-specific statistics. For information on different debug interfaces, see the appropriate documentation, for example:

- *RealView ICE User Guide*
- *Multi-ICE® User Guide*
- *ARM MultiTrace™ User Guide*.

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

- *Displaying register contents*
- *Formatting options* on page 6-4
- *Changing register contents* on page 6-6
- *Viewing different registers* on page 6-7
- *Understanding the register view* on page 6-9
- *Viewing debugger internals* on page 6-11
- *Defining new registers* on page 6-18
- *Interactive operations* on page 6-18.

6.1.1 Displaying register contents

To examine the contents of registers:

1. Select **View** → **Registers** to display the Register pane and bring the **Core** tab to the front.
2. Connect to your target and load an image, for example `dhrystone.axf`.
3. Select **Edit** → **Advanced** → **Show Line Numbers** to display line numbers.

   This is not necessary but might help you to follow the examples.
4. Click on the **Src** tab to view the source file `dhry_1.c`.
5. Set a default breakpoint by double-clicking on line 301.
6. Click **Run** to start execution.

7. Enter `5000` when asked for the number of runs.

   The program starts and then stops when execution reaches the breakpoint at line 301. The red box marks the location of the PC when execution stops.

8. The contents of the Register pane are updated to show the program status as the target stops, shown in Figure 6-1.

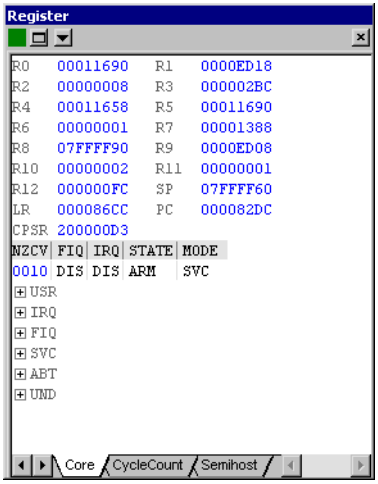

Figure 6-1 Register pane

9. Click on the **Core** tab to view base registers for your target processor.

   The Register pane displays tabs appropriate to the target processor running your image and the target interface. Different target processors contain different registers and so the contents of this pane change depending on the target you are debugging.

   For more details on the contents of this pane see:

   - *Viewing different registers* on page 6-7 for examples of other tabs.
   - *Understanding the register view* on page 6-9 for information on registers.
   - *Viewing debugger internals* on page 6-11 for information on debugger internals and statistics.

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

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

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

The first time you perform this operation in a debugging session, a new instance of the Memory pane is displayed as a floating pane. Subsequent **View Memory At Value** operations use this instance of the Memory pane to display the memory contents, even if you have docked that pane.

If you change the docked instance of the Memory pane to a different pane view (such as Process Control), and you later perform the **View Memory At Value** operation, the docked Memory pane is redisplayed to the right of the changed pane view (Process Control in this example).

This Memory pane is also used if you perform the view memory operations from the Watch pane or the File Editor pane.

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

14. Double-click on the red marker disc to clear the breakpoint at line 301.

Note

If you are using *RealView ARMulator® ISS* (RVISS), as in this example, 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
- ARM720T™ r4, does not distinguish trace PID and FCSE r13.

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.

6.1.2 Formatting options

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

Global formatting of all registers

Click the **Pane** menu to select formatting options for all registers currently displayed. This applies formatting to all registers, except those registers that you have specifically formatted on an individual basis. For example, you might want to display contents as values rather than enumerations, or to display values in decimal format. This change applies to all tabs in the Register pane.

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

For selected registers

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

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

The values of the `FIQ`, `IRQ`, `STATE`, and `MODE` fields of the CPSR register can be set using names that enumerate to numerical values. The options offered on this menu depend on the register currently under the mouse pointer. Position your pointer over the `MODE` field value of the CPSR register, which is set to `SVC` by default, and then right-click to display the status register context menu.

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

Select **Set Enumeration...** to display the selection box shown in Figure 6-2.

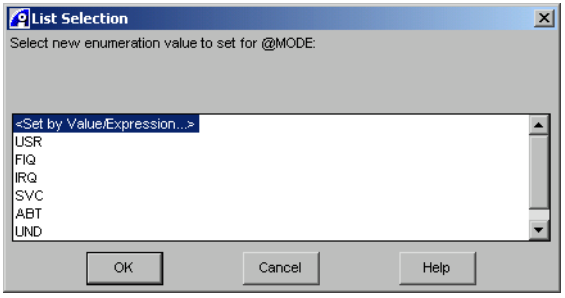

Figure 6-2 Setting enumeration values

This dialog enables you to select a new value for the chosen register or to use an expression to define the contents. For example, highlight the entry `<Set by Value/Expression...>`. Click **OK** to display a box where you can enter the value or choose from a drop-down list.

As you change a value in a CPSR register field, the CPSR register value is also updated.

6.1.3 Changing register contents

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

- Set the contents of a register by pasting variables from other windows. For example, right-click on a value in the source-level view and select **Copy** from the context menu. Right-click in the register whose value you want to change and select **Paste Value** from the context menu.
- If you want to use a favorite data value that you have saved, right-click on a register and select **Set from Favorites** from the context menu to display a Favorites Chooser/Editor dialog box (see *Using the Favorites Chooser/Editor dialog box* on page 8-29).
- Highlight a value in the **Src** tab in the File Editor pane and then use drag-and-drop to copy the expression into the contents of a chosen register in the Register pane. See *Dragging and dropping expressions into register contents* for an example.
- Select **Debug → Memory/Register Operations → Set Register...** from the Code window main menu to display the Interactive Register Setting dialog box.

Any register contents you change are displayed in blue.

Note

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

Dragging and dropping expressions into register contents

You can drag and drop an expression from a source file into the contents of a register function name. The following example shows how to you can set the PC register to the address of a function:

1. Load the `dhrystone.axf` image.
2. Enable line numbering and locate line 149 in the `dhry_1.c` source.

3. Double-click on the `Proc_5` function name.

4. Click on the function name, and drag it to the PC register.

   The address of the `Proc_5` function is loaded into the PC register, and the source level view changes to show the code for the `Proc_5` function. The **Dsm** tab is also updated.

   Note

   If the function code is in a different source file, and that source file is not open, RealView Debugger cannot determine the context, and informs you of this in the **Src** tab. However, the **Dsm** tab is updated.

Updating register contents

The Register pane updates automatically when register contents change. This is set by default, as indicated by the checked option **Automatic Update** on the **Pane** menu.

If you change this default option, you must update the pane display manually. Select **Update Window** from the Register pane context menu to update the view. RealView Debugger updates the pane contents and displays changed values in blue for improved readability.

6.1.4 Viewing different registers

The Register pane displays tabs and registers appropriate to your:

- target processor, that is different target processors contain different registers and so the contents of this pane change depending on the target you are debugging
- target interface, for example RealView ICE
- target configuration, for example if you are using a `.bcd` file, the pane includes extra tabs to provide extended target visibility.

In Figure 6-3 on page 6-8, the Register pane shows registers for an ARM940T core and the ARM Integrator™/AP board using Multi-ICE.

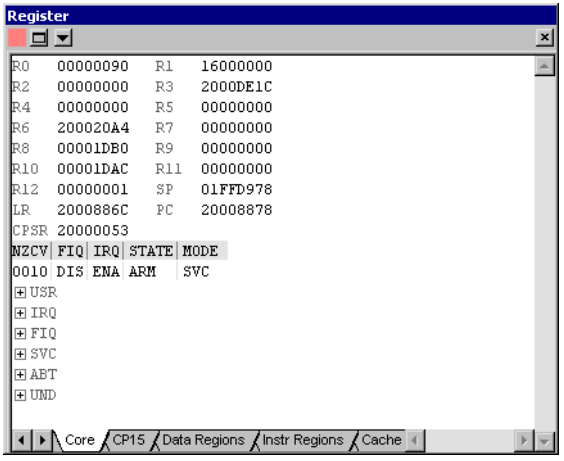

Figure 6-3 Registers for an ARM940T debug target

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

In the example in Figure 6-4, the Register pane displays register contents for a CEVA-Oak debug target.

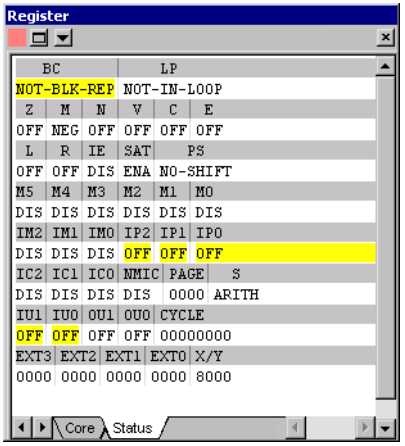

Figure 6-4 Registers for a CEVA-Oak debug target

Here you can see the **Status** tab displaying the status registers.

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.

For details on setting up multiple Code windows and attaching to different debug targets, see the multiprocessing chapter in *RealView Debugger v1.8 Extensions User Guide*.

6.1.5 Understanding the register view

ARM processors support up to seven processor modes depending on the architecture version, for example User (`USR`), Supervisor (`SVC`), and `FIQ`. All modes, except User mode, are referred to as *privileged* modes.

ARM processors have thirty-seven registers arranged in partially overlapping banks. There is a different register bank for each processor mode, for example `USER` or `FIQ`. Using banked registers gives rapid context switching for dealing with processor exceptions and privileged operations.

RealView Debugger displays registers in named groups to reflect how registers are banked, for example `USR`, `IRQ`, and `FIQ` for an ARM7TDMI® core. Click on the plus, or minus, sign to expand, or collapse, the view (shown in Figure 6-5 on page 6-10).

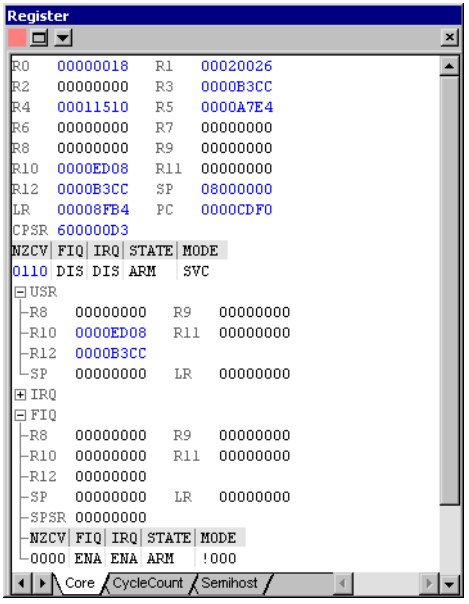

Figure 6-5 Viewing registers

At first sight, it might appear that some registers are missing or that extra registers are visible for different processor modes. For example, FIQ contains R8, R9, R10, R11, R12, SP, LR, and SPSR. These are the registers in the bank for that processor mode that are banked out when an FIQ exception occurs.

However, USR also contains R8, R9, R10, R11, R12, SP, and LR. These are banked out registers indirectly accessible with LDM and STM instructions of the form:

```
LDM rX, (r8-r14)^
```

```
STM rX, (r8-r14)^
```

These examples use the ^ suffix to specify that data is transferred into or out of the USR mode registers. The register list must not contain the PC.

You must load banked out registers from memory to modify them, and you must store them to memory to read them. However, they might be of interest when writing task context switch code or a coprocessor emulator.

For full details on LDM and STM commands see the appropriate documentation, for example:

- *RealView Compilation Tools Assembler Guide*
- *RealView Compilation Tools Developer Guide.*

6.1.6 Viewing debugger internals

Depending on your debug target, debugger internals appear in tabs in the Register pane:

- *Viewing RVISS cycle counts*
- *Viewing internal variables* on page 6-12
- *Viewing semihosting controls* on page 6-13.

Viewing RVISS cycle counts

If you are using RVISS to simulate a target processor, and connect to RVISS using the RealView Connection Broker interface, the **CycleCount** tab displays a range of internal cycle count variables. What is shown depends on the processor that you have chosen to simulate, for example Figure 6-6 shows the display for an ARM7TDMI core.

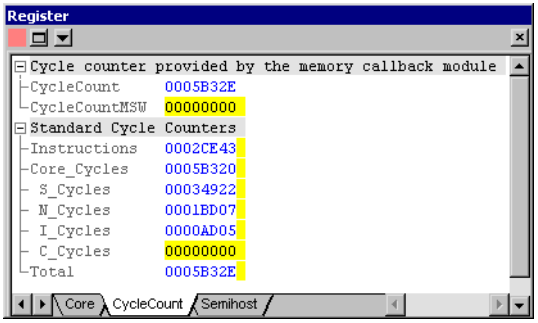

Figure 6-6 Viewing CycleCount tab (RVISS)

Figure 6-6 shows:

Instructions The number of program instructions executed.

Core_Cycles

Internal core cycles indicating the time an instruction spends in the execute stage of the pipeline.

S_Cycles The number of sequential cycles performed. The CPU requests transfer to, or from, the same address, or an address that is a word or halfword after the address used in the preceding cycle.

N_Cycles The number of nonsequential cycles performed. The CPU requests transfer to, or from, the same address, or an address that is unrelated to the address used in the preceding cycle.

I_Cycles The number of internal cycles performed. The CPU does not require a transfer because it is performing an internal function (or running from cache).

C_Cycles The number of coprocessor cycles performed.

Total The sum of the `S_Cycles`, `N_Cycles`, `I_Cycles`, and `C_Cycles`.

For information on what is shown for different target processors, see:

- the section that describes RVISS cycle types in the *RealView ARMulator ISS User Guide*
- the hardware documentation for the target processor you are simulating.

Viewing internal variables

RealView Debugger uses internal variables like any other program. These are set up as part of the target device configuration or the BCD file configuration. The variables it uses depend on your debug target. If you are using RealView ICE, Multi-ICE, or the RDI interface of RVISS, you see a different set of debugger internals in the **Debug** tab. An example of the **Debug** tab for RealView ICE is shown in Figure 6-7 on page 6-13.

Note

The RDI connection to RVISS in RealView Debugger is deprecated in this release.

Note

It is strongly recommended that you do not change the semihosting ARM and Thumb® SWI numbers. If you do, you must:

- change all the code in your system, including library code, to use the new SWI number
- change the `ARM_SWI` and `Thumb_SWI` settings to use the new SWI numbers.

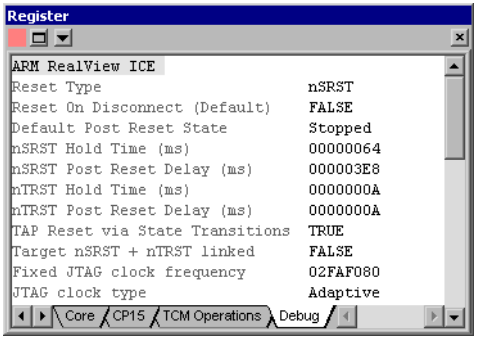

Figure 6-7 Viewing internal variables (RealView ICE)

For information on what is shown for different target interfaces, see the appropriate documentation:

- *RealView ICE User Guide*
- *Multi-ICE User Guide*
- the hardware documentation for your target processor.

Viewing semihosting controls

Semihosting enables code running on an ARM architecture-based target to use facilities on a host computer that is running RealView Debugger. Examples of such facilities include the keyboard input, screen output, and file system I/O. The method of controlling semihosting depends on what connection you are using:

- *RVISS using RealView Connection Broker interface*
- *JTAG connection using RealView ICE or Multi-ICE* on page 6-15.

RVISS using RealView Connection Broker interface

If you connect to RVISS using the RealView Connection Broker interface, the **Semihost** tab in the Register pane, shown in Figure 6-8 on page 6-14, enables you to control the behavior of semihosting.

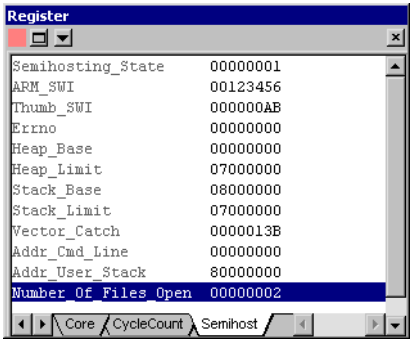

Figure 6-8 Viewing Semihost tab (RVISS)

Figure 6-8 shows:

Semihosting_State

Indicates the semihosting state:

1 Semihosting on

0 Semihosting off

ARM_SWI The SWI used for semihosting in ARM state.

Thumb_SWI

The SWI used for semihosting in Thumb state.

Note

It is strongly recommended that you do not change the semihosting ARM and Thumb SWI numbers. If you do, you must:

- change all the code in your system, including library code, to use the new SWI number
- change the `ARM_SWI` and `Thumb_SWI` settings to use the new SWI numbers.

Errno The `errno` value of the last SWI operation manipulating files, returned by the `SYS_ERRNO` SWI.

Heap_Base The lowest address of the system heap, returned by the `SYS_HEAPINFO` SWI.

Heap_Limit The highest address available for the system heap, returned by the `SYS_HEAPINFO` SWI.

Stack_Base The lowest address of the system stack, returned by the SYS_HEAPINFO SWI.

Stack_Limit The highest address available for the system stack, returned by the SYS_HEAPINFO SWI.

Vector_Catch

A bit mask of the current vector catch, showing the SWI processor exceptions that are currently enabled. It is suggested that you do not modify this in the Register pane. Instead, enable or disable the processor exceptions using the following option on the main menu:

Debug → Processor Exceptions...

Addr_Cmd_Line

The address of the command line in ARM space that is to receive the command line to the image (SWI_GetEnv).

Addr_User_Stack

The address of the user stack (SWI_GetEnv).

Number_Of_Files_Open

The number of files that the semihosting operations currently have open.

JTAG connection using RealView ICE or Multi-ICE

When you are connected via RealView ICE or Multi-ICE, if standard semihosting is enabled, then the target processor enters debug state when a semihosting operation is performed. To enable semihosting you must set the variable semihost_enabled, shown in Figure 6-7 on page 6-13 and Figure 6-9 on page 6-16:

- Standard semihosting is enabled by default, and is the only mode available on RealView ICE. The behavior of semihosting depends on your target core, and your target vehicle (see *Standard semihosting behavior* on page 6-16)
- DCC semihosting is available on Multi-ICE (see *DCC semihosting* on page 6-17).

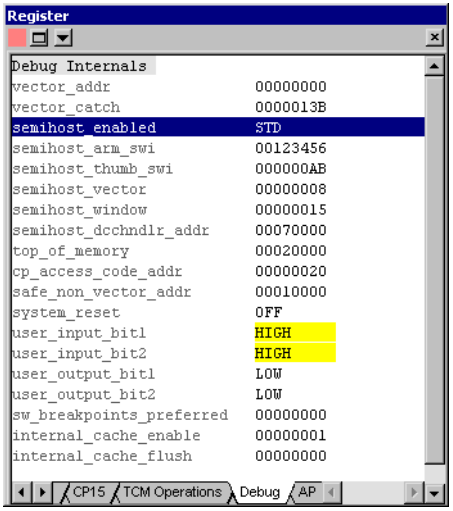

Figure 6-9 Specifying semihosting (Multi-ICE)

Standard semihosting behavior

If there is ROM or FLASH at the SWI Vector, then the use of semihosting requires a hardware breakpoint on certain cores, such as ARM7TDMI. In this case, there might be insufficient hardware breakpoint resources left to permit single instruction stepping or source level stepping, so it might be necessary to disable semihosting.

The behavior also depends on the target vehicle you are using to connect to the core. For example, if you connect to an ARM7TDMI on an Integrator/AP board using RealView ICE, the following warning messages are displayed:

- for software breakpoint modes `AUTO`, `WATCHPOINT`, and `BKPT`:

  ```
  Warning: A software breakpoint is being used to simulate reset vector
  catch. This may fail to be hit if the memory is remapped when a reset
  occurs.
  ```

- for software breakpoint mode `NONE`:

  ```
  Warning: Insufficient hardware resources to enable requested vector catch
  events. Some vector catch events have been disabled.
  ```

See the *RealView ICE User Guide* for details on setting the software breakpoint mode. Also see *Breakpoints and memory map locations* on page 4-9 for more details on breakpoints and memory mapping.

DCC semihosting

If you are using Multi-ICE to connect to a target, DCC semihosting is supported in addition to standard semihosting. DCC semihosting is not supported if you are using RealView ICE.

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 RealView ICE or Multi-ICE interface, without stopping the program or entering debug state.

DCC is a word-size communications mechanism implemented in the debug part of the ARM core as a number of registers in CP14. DCC takes the form of a 32-bit register through which data might be communicated between code running on the host and code running on the target.

Note

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

When using DCC semihosting, 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.

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

Specify DCC semihosting by setting the variable `semihost_enabled` to **DCC** (see Figure 6-9 on page 6-16). To do this, right-click on the `semihost_enabled` variable, and select **DCC** from the context menu. 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 *Multi-ICE User Guide*.

6.1.7 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 that describes configuring custom targets in *RealView Debugger v1.8 Target Configuration Guide* for details.

6.1.8 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 6-21
- *Operating on memory contents* on page 6-25
- *Data width on memory accesses* on page 6-27
- *Changing memory contents* on page 6-28
- *Managing multiple targets* on page 6-29
- *Interactive operations* on page 6-29.

6.2.1 Displaying memory contents

To examine the contents of memory:

1. Select **Target → Reload Image to Target** to reload the image `dhrystone.axf`.

   You can also reload an image using the **Reload Image** button on the Image toolbar.

2. Click on the **Src** tab to view the source file `dhry_1.c`.

3. Set a default breakpoint by double-clicking on line 301.

4. Click **Run** to start execution.

5. Enter `5000` when asked for the number of runs.

   The program starts and then stops when execution reaches the breakpoint at line 301. The red box marks the location of the PC when execution stops.

6. If no Memory pane is visible, select **View → Memory** to display the Memory pane, shown in Figure 6-10 on page 6-20.

   You can set start addresses using in-place editing or using the context menu.

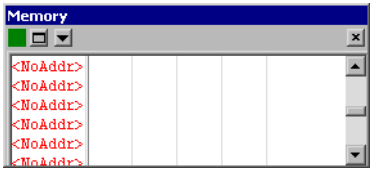

Figure 6-10 Memory pane

7. Right-click in the address column in the Memory pane to display the **Address** context menu.

8. Select **Set Start Address...** to display the dialog box shown in Figure 6-11.

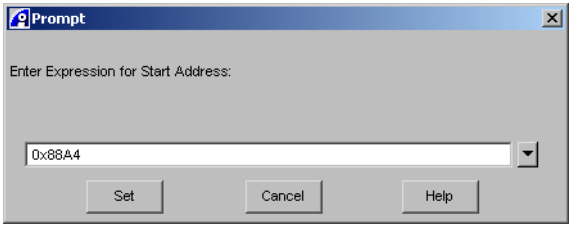

Figure 6-11 Memory start address selection box

   You specify the start address by giving an address in hexadecimal or by giving a C/C++ expression that RealView Debugger computes to obtain the starting address. You can use any valid expression using constants and symbols.

   You can also use the drop-down arrow to select an expression from a browser or to re-use a value entered previously. The drop-down also gives access to your list of personal favorites (see *Using the Favorites Chooser/Editor dialog box* on page 8-29) where you can store a memory address for re-use in this, or future, debugging sessions.

   In this example, memory addresses of interest are in the region of `000088A0` so set the start address to examine memory from this location.

   Numbers entered here must start with a zero. This means that RealView Debugger can distinguish these entries from valid variable names.

9. Enter the required location, for example `0x00088A4`, and then click **Set** to update the Memory pane.

   The memory display is arranged in columns. The left-most column shows the memory address. The memory contents are shown in the other columns. The number of columns displayed varies depending on the size of the pane. Color coding is used to distinguish the type of memory being displayed, see *Display colors* on page 6-24 for details.

10. Monitor changes in the memory display as you step through your program.

11. Double-click on the red marker disc to clear the breakpoint at line 301.

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

If you have unselected the option **Automatic Update**, you can use this option to update the memory display manually. You can update the display using this option at any time when execution is stopped. This enables you to catch any memory updates made externally.

Set Start Address...

Select this option to enter a C/C++ expression to compute the start address. This displays the selection box shown in Figure 6-11 on page 6-20 with the current expression already displayed. Change the address and click **Set** to specify the start address for the memory view.

Previous Start Address

Uses a previous start address for displaying the contents of memory.

The history list holds up to 16 previous start addresses added when:

- you enter a new start address or expression
- the current expression is recomputed to generate a new start address
- the start address is set from the **Address** context menu.

Re-evaluate Start Address

Where you have used a C/C++ expression to compute the start address, select this option to recompute the expression and, where necessary, start at the new location. Where you have used a fixed value to specify the start address, select this option to update only the pane contents.

Set Number of Columns to show...

When the Memory pane first opens, the number of columns you can see depends on the size of the pane and is chosen so as to show an even number of bytes. Use this option to change the number of columns visible in the display. Use the selection box to show up to 32 columns in a single

window. This number does not include the column used when the ASCII display option is selected, see the description of **Show ASCII** in *Data sizes* on page 6-23.

The default setting is 0 to configure the number of columns to fit the window size.

Automatic Update

Updates the memory display automatically, that is when:

- you change memory from anywhere in RealView Debugger
- program execution stops.

This is the default.

Re-evaluate Start Address on Update

Where you have used a C/C++ expression to compute the start address, select this option to recompute the expression when the pane contents are updated (see the description of **Automatic Update**), and start at the new computed location where necessary.

Timed Update when Running

If you are using RealMonitor or an RTOS extension, 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 by the **Timed Update Period** option.

This option is enabled when:

- RealMonitor is activated (see the chapter that describes configuring custom targets in *RealView Debugger v1.8 Target Configuration Guide* for details)
- you are in RSD mode and where supported by the underlying debug target (see the chapter that describes RTOS support in *RealView Debugger v1.8 Extensions User Guide* for details).

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.

Size Displays a submenu where you can select various data sizes (see *Data sizes* on page 6-23).

Format

Displays a submenu where you can select various data formats (see *Data formats*).

Data sizes

The **Pane** menu contains a **Size** submenu to define how data values are displayed in the Memory pane. The display size used for viewing memory contents varies depending on the data types supported by your target processor:

Signed Decimal

Displays the memory contents as negative or positive values where the maximum absolute value is half the maximum unsigned decimal value.

Unsigned Decimal

Displays the memory contents from 0 up to the highest value that can be stored in the number of bits available.

Hexadecimal

This displays memory contents as hexadecimal numbers.

Hexadecimal with leading Zeros

Displays memory values in hexadecimal format including leading zeros.

This is the default display format for data values in this pane.

Show ASCII Adds another column to the Memory pane, on the right hand side, to show the ASCII value of the memory contents.

ASCII format displays column values as characters. The ASCII format is useful if, for example, you are examining the copying of strings and character arrays by transfer in and out of registers.

Any non-printable value is represented by a period (.).

Data formats

The **Pane** menu contains a **Format** submenu to define how data values are displayed in the Memory pane. The display format used for viewing memory contents varies depending on the data types supported by your target processor:

Minimum Access Size

Displays memory contents in the format specified as the minimum memory access size for the target. This is the default.

Bytes (8 bits) Each column displays 8 bits of data.

Half Words (16 bits)

Each column displays 16 bits of data. Where your debug target is an ARM processor halfwords are aligned on 2-byte boundaries.

Words (32 bits)

Each column displays 32 bits of data. Where your debug target is an ARM processor long words are aligned on 4-byte boundaries.

Double Words (64 bits)

Each column displays 64 bits of data.

Fixed (word size)

Enables you to use fixed point format for displaying numeric values, that is based on the natural size for the debug target processor. The default format is unsigned and one less than the number of bits in the value.

Fixed... Displays a selection box that enables you to specify a fixed point format to display numeric values. The value entered here becomes the default display format for the pane.

Floats (32 bits)

Displays values in floating point IEEE format, occupying four bytes, for example:

```
2.5579302e-041
```

Doubles (64 bits)

Displays values in floating point IEEE format, occupying eight bytes, for example:

```
4.71983561663e+164
```

Display colors

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

- Black specifies RAM or memory that can be modified.
- Blue shows those contents that have changed since the last update. Light blue indicates a previous update.
- Yellow indicates the contents of ROM.
- Green indicates Flash memory known to RealView Debugger. Otherwise the values are displayed in yellow, indicating ROM.

Note

If the Flash memory locations are shown with a yellow background in the Memory pane, then the `Flash_type` setting for the Flash memory area in the BCD file is either pointing to an invalid FME file, or is not set.

- Red **** indicates one of:
  - no memory is defined at this location
  - memory at this location is defined as Auto meaning it is determined when loading your application program
  - memory is defined as *prompt* meaning that you are prompted to confirm the usage when loading the application.
- Red !!!! indicates that there has been a failure in performing the memory operation. Double-click, with the right mouse button, at this location to get an explanation of the problem.

6.2.3 Operating on memory contents

You can perform different operations on the memory displayed in the Memory pane using the context menus. The menu shown, and the options available, depend on the type of memory under the cursor when you right-click and on the valid licenses that you have. The color-coded display helps you to identify the memory type. For a description of the context menus, see:

- *Address context menu* on page 6-26
- *Memory value context menu* on page 6-26.

Note

If you right-click on a memory cell to access a context menu, any change you specify is made to the cell under the cursor. This is independent of any highlighted cells in the view.

If you double-click on an address in the address column of the Memory pane, an edit field is displayed. You can enter a specific address, or a symbol that specifies an address, in this field. For example, to display memory at the address pointed to by the SP register, enter `@SP`.

Address context menu

Right-click on a memory address to display the **Address** context menu that provides options to set the start address:

Update

Updates the display in the Memory pane.

Set Start Address...

Enables you to specify the starting address for the display of memory contents.

Previous Start Address

Enables you to use the previous starting address for the display of memory contents.

Re-evaluate Start Address

Where you have used a C expression to compute the start address, select this option to recompute the expression and, where necessary, start the display at the new location.

Memory value context menu

Right-click on a memory cell (or byte) shown as black or green, that is where the type is ROM, Flash, or modifiable, to display the **Memory value** context menu. This context menu contains the following options:

Update

Updates the display in the Memory pane.

Set Start Address from Content

Enables you to use the cell contents as the starting address for the display of memory contents. This option is enabled when the cell contains a scalar the size of an address or pointer.

Show Symbol from Content

RealView Debugger looks up the address held in the cell and displays any symbol at that address.

Show Symbol at Address

Displays any symbol at the address contained in the cell, and not the contents.

Set to 0 Enables you to set the current memory cell to zero.

Increment Enables you to add 1 to the contents of the memory cell.

Decrement Enables you to subtract 1 from the contents of the memory cell.

Set Value... Displays a prompt where you can enter a new value for the memory cell. This new value is then entered and the memory display is updated.

A memory cell can also be changed using in-place editing.

Set Memory Interactive...

Enables you to use memory interactive operations available in RealView Debugger.

Fill Memory with Pattern...

Enables you to fill memory starting at this location.

Set Break At...

Displays the Set Address/Data Break/Tracepoint dialog box where you can specify a breakpoint on the current memory cell. The type of breakpoint offered depends on the type of memory at the chosen location. For example, if the memory is defined as ROM, RealView Debugger offers a hardware breakpoint first.

Set Tracepoint...

Enables you to set tracepoints based on the current memory view.

Right-click on a memory cell, or byte, that is yellow, that is where the type is ROM, to see the ROM-specific context menu that offers a subset of these options.

Memory errors

Where a memory cell contains !!!! (colored red), this shows that there has been an error in the memory operation. Right-click to display a menu with a single option:

Show Error Code...

Select this to display the error code returned from the debug target when the memory operation failed.

6.2.4 Data width on memory accesses

By default, RealView Debugger makes word-size accesses when reading or writing memory. You can change this in your memory map or by specifying the `Memory_block` settings in the `Advanced_Information` block in your target configuration file. For details see the chapter that describes configuring custom targets in *RealView Debugger v1.8 Target Configuration Guide*.

6.2.5 Changing memory contents

To change memory contents:

1. Select **Target → Reload Image to Target** to reload the image `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Set a default breakpoint by double-clicking on line 301.
4. Click **Run** to start execution.
5. Enter `5000` when asked for the number of runs.

   The program starts and then stops when execution reaches the breakpoint at line 301. The red box marks the location of the PC when execution stops.
6. Use the **Pane** menu, in the Memory pane, to set the start address to `0x89A2` and display the ASCII values, shown in Figure 6-12.

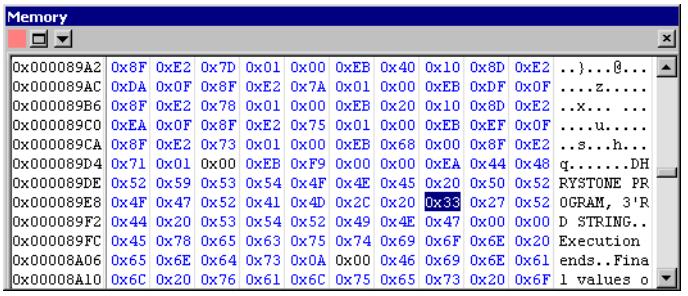

Figure 6-12 Example memory display

The memory locations `000089EF-000089F2` contain the four hexadecimal values `0x33`, `0x27`, `0x52`, and `0x44` corresponding to the string `3'RD`.

Right-click in the value at `000089ef`, that is `0x33`, to display the **Memory value** context menu (see *Operating on memory contents* on page 6-25).

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 6-11 on page 6-20, where you can enter the new value.

Enter the required hexadecimal value `0x4E`, or enter `'N'`, and click **Set** to update the memory display. You can use uppercase or lowercase to enter the new value.

The new value is displayed in blue and the ASCII value changes from `3` to `N`.

You can also use the drop-down arrow to select from a browser or to re-use a value entered previously. The drop-down also gives access to your list of personal favorites where you can store a data value for re-use in this, or future, debugging sessions.

7. Change the value at location `000089F0` from `0x27` to `0x6F` (lowercase o).

8. Data values can be entered in a format that is different from the display format. Right-click in the location `000089F1` (`0x52`), for example, enter the decimal value `32` and then click **Set**. The memory pane is updated with the new value, a space, displayed in the chosen display format.

9. You can also use in-place editing to change memory contents. Double-click in the value `0x44` (`D`) at location `000089F2` and change it to `0x32`. Press Enter to confirm the new value.

   If you press Escape then any changes you made in the highlighted field are ignored.

10. View the changed values in the memory display. Each new value is displayed in blue as the pane contents are updated.

11. View the changes you have made in the messages displayed when your program completes. The string "`3'RD`" has been replaced by "`No 2`".

12. Double-click on the red marker disc to clear the breakpoint at line 301.

6.2.6 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.

For details on setting up multiple Code windows and attaching to different debug targets, see the multiprocessing chapter in *RealView Debugger v1.8 Extensions User Guide*.

6.2.7 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

The stack, or run-time stack, is an area of memory used to store function return information and local variables.

Executing a function sets up the stack. As the new function is called, a record is created on the stack including traceback details, and local variables. At this point these arguments and local variables become available to RealView Debugger, and you can access them through the Code window.

When the function returns, the area of the stack occupied by that function is recovered automatically and can then be used for the next function call.

In a typical memory-managed ARM processor, the memory model comprises:

- a large area of application memory starting at the lowest address (code and static data)
- an area of memory used to satisfy program requests, the heap, that grows upwards from the top of the application space
- a dynamic area of memory for the stack which grows downwards from the top of memory.

The *Stack Pointe*r (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:

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 **Target → Reload Image to Target** to reload the image `dhrystone.axf`.
2. Select **View → Stack** to view the Stack pane.
3. Click on the **Src** tab to view the source file `dhry_1.c`.
4. Set a default breakpoint by double-clicking on line 301.
5. Click **Run** to start execution.
6. Enter `5000` when asked for the number of runs.

   The program starts and then stops when execution reaches the breakpoint at line 301. The red box marks the location of the PC when execution stops.
7. View the updated Stack pane, shown in Figure 6-13.

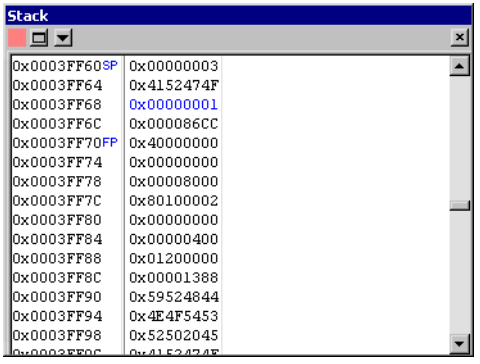

Figure 6-13 Viewing the stack

The stack pointer, marked by `SP`, is located at the bottom of the stack. The frame pointer, marked by `FP`, shows the starting point for the storage of local variables.

8. Monitor changes in the Stack pane as you step through your program, for example by clicking **Hi-level Step Into**.

9. Double-click on the red marker disc to clear the breakpoint at line 301.

The stack is displayed in columns:

Address The left column contains the memory addresses of the stack.

In some target processors that use a Harvard architecture, for example a DSP, a `D` is appended to show that this is a data address. You must include this letter when specifying such an address as the starting address.

Value The right column displays the contents of the addresses in the stack.

As with the Memory pane, the memory display in the Stack pane is color-coded for easy viewing and to enable you to monitor changes.

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 to manage the display of stack contents during your debugging session, change the display format, and extract data from the pane for use in other panes or windows. Highlight an entry in the display and then choose from the list of available options:

Copy Copies the chosen entry from the list to the clipboard.

Previous Start Address

Enables you to use the previous starting address for the display of memory contents.

Signed Decimal

Displays the memory contents as negative or positive values where the maximum absolute value is half the maximum unsigned decimal value.

Unsigned Decimal

Displays the memory contents from 0 up to the highest value that can be stored in the number of bits available.

Hexadecimal

Displays memory contents as hexadecimal numbers.

Hexadecimal with leading Zeros

Displays memory values in hexadecimal format including leading zeros.

This is the default display format for data values in this pane.

Show ASCII Adds another column to the Stack pane, on the right hand side, to show the ASCII value of the memory contents.

ASCII format displays column values as characters. The ASCII format is useful if, for example, you are examining the copying of strings and character arrays by transfer in and out of registers.

Any non-printable value is represented by a period (.).

Data formats

The **Pane** menu contains an extended panel to define how data values are displayed in the Stack pane. The display format used for viewing memory contents varies depending on the data types supported by your target processor:

Bytes (8 bits) Each column displays 8 bits of data.

Half Words (16 bits)

Each column displays 16 bits of data. Where your debug target is an ARM processor halfwords are aligned on 2-byte boundaries.

Words (32 bits)

Each column displays 32 bits of data. Where your debug target is an ARM processor long words are aligned on 4-byte boundaries.

Double Words (64 bits)

Each column displays 64 bits of data.

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 6-25.

Changing the stack pointer

As you step through your code, the default stack pointer is used, shown in Figure 6-13 on page 6-31. You can specify an expression or a register to use as the stack pointer from the Stack pane:

1. Right-click on a stack address or the Stack pane background to display the **Stack** context menu.

   Note

   If you right-click on a stack value, the **Stack Value** context menu is displayed.

2. Select **Set Start Address...** to display the address prompt box.
3. Enter an expression or a register, for example `@R9`, as the new stack pointer.

   You can also use the drop-down arrow to select an expression from a browser or to re-use a value entered previously. The drop-down also gives access to your list of personal favorites (see *Using the Favorites Chooser/Editor dialog box* on page 8-29) where you can store a memory address for re-use in this, or future, debugging sessions.
4. Click **Set** to confirm your choice and close the address prompt box.

The new stack pointer is marked by *Expression Pointer* (`EP`), located at the bottom of the stack.

If you enter a blank expression, or remove the existing expression, in the address prompt box, RealView Debugger reverts to using the default stack pointer register. In this example, this was `R13` shown in Figure 6-1 on page 6-3.

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 stack value to display the **Stack Value** context menu.
2. Select **Set Break At...** from the **Stack Value** context menu.
3. Complete the entries in the Set Address/Data Breakpoint dialog box.
4. Click **OK** to close the dialog box and set the breakpoint.

RealView Debugger sets a breakpoint on a symbol address where it exists on the stack. As soon as you exit the function, the address is no longer meaningful. Do not, therefore, use such a breakpoint where execution runs past the function return call.

Unlike the Watch pane, the Stack pane acts like a snapshot of a chosen address. It does not track each invocation of a function and so is not able to track the chosen symbol.

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 call stack. Similarly, as a function completes execution and returns control normally, it is removed from the call stack. The call stack, therefore, contains details of all functions that have been called but have not yet completed execution.

RealView Debugger includes features that enable you to monitor variables and access traceback as your debugging session develops:

- *Using the Stack pane*
- *Using the Call Stack pane*.

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 6-30 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. You can also use the Call Stack pane to set breakpoints.

The Call Stack pane shows the:

- name of the function
- line number in the source file from which the function was called
- parameters to the function.

To use traceback:

1. Select **Target → Reload Image to Target** to reload the image `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Set an unconditional software breakpoint by double-clicking on line 301.
4. Click **Run** to start execution.
5. Enter `5000` when asked for the number of runs.

The program starts and then stops when execution reaches the breakpoint at line 301. The red box marks the location of the PC when execution stops.

6. Select **View** → **Call Stack** to view the Call Stack pane, shown in Figure 6-14.

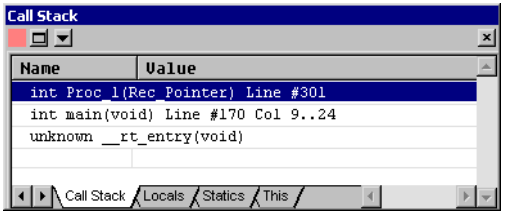

Figure 6-14 Multistatement details in the Call Stack pane

7. Continue to step through your program. If the current line is a multistatement line, the Call Stack pane shows the information in the form of line and column details, shown in Figure 6-14.
8. Monitor changes in the Call Stack pane as you step through your program.
9. Double-click on the red marker disc to clear the breakpoint at line 301.

Tabs in the Call Stack pane

The Call Stack pane contains the tabs:

Call Stack Displays details of the functions currently on the stack, shown in Figure 6-14.

Locals Displays a list of the variables that are local to the current function.

Statics Displays a list of static variables local to the current module.

This In C++ the `this` pointer locates the object for which the member function was called. It is C++ specific.

Using the Pane menu

Click on the **Pane** menu button to display the **Pane** menu that contains options to:

- manage variables
- change the display format
- change how contents are updated
- extract data from the Call Stack pane for use in other panes or windows.

Click on the **Call Stack** tab, highlight an entry in the functions list and display the **Pane** menu options:

Copy Copy the chosen entry, where enabled.

Timed Update when Running

Not available in this release.

Timed Update Period...

Not available in this release.

Update Window

Updates the contents of the Call Stack pane. Use this when **Automatic Update** is disabled.

Automatic Update

Refreshes the Call Stack pane as soon as a watchpoint is triggered and execution stops. This is enabled by default.

Show char* and char[] as strings

Displays local variables of type `char*` and `char[]` (or casted) as strings.

This is enabled by default.

Show integers in hex

Displays all integers in hexadecimal format. Disabling this option displays all integers in decimal. This is enabled by default.

Properties... Displays a text description of the item under the cursor.

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.

Using the Function menu

Right-click on a function in the **Call Stack** tab to display the **Function** context menu:

Scope To Scopes to the chosen function.

Break at... Sets a breakpoint at the address defined by the chosen function, if this is permitted.

BreakIf... Displays the Breakpoint type selection box.

Go to This continues execution until the specified point in the stack is reached.

Properties... Displays a text description of the item under the cursor.

Using the Variable menu

Right-click on a chosen entry `Name` or `Value` in the **Locals** tab to display the **Variable** menu. This context menu provides options that operate on selected variables only:

Update Updates the displayed value for the chosen expression. This is applied in combination with any update options you set from the **Pane** menu.

Format... Displays a selection box where you can highlight the required format for the expression from the list of available formats.

If a variable contains multibyte characters, you can choose to view it using `ASCII`, `UTF-8`, or `Locale` encoding.

6.4.4 Stack controls

When working with the Call Stack pane there are two **Context** controls available from the Debug toolbar:

- Stack up
- Stack down.

See *Context controls* on page 6-34 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 6-42
- *Saving watches as favorites* on page 6-46.

6.5.1 Setting watches in source-level view

To set a watch:

1. Select **Target → Reload Image to Target** to reload the image `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Right-click on a variable name, for example `Run_Index` at line 146, and select **Watch** from the **Source Variable Name** menu.
4. Set a second watch, for example on the variable `Enum_Loc` at line 155.

If you are working in source-level view and the Watch pane is visible, you can set watches in other ways:

- use the option **Enter New Expression...** from the **Pane** menu in the Watch pane
- drag-and-drop the variable to watch
- select a variable to watch, copy it, and paste it into the Watch pane.

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 → Watch** to view the Watch pane, shown in Figure 6-15 on page 6-42.

The Watch pane contains a series of tabs. The **Watch1** tab is selected by default. You can set expressions on different tabs to make it easier to manage what is being watched during your debugging session. Click on the required tab to select it.

Expressions are listed in the order they were created. You can drag the column headings to display the full name or value if required. Where an entry contains subentries, for example an array, a plus sign is appended to the name. Click on this to expand the display.

Using the Pane menu

Highlight an entry in the watched variables list and click the **Pane** menu button to display the **Pane** menu. This contains:

Copy Select this option to copy the chosen value from the list to the clipboard. From here it can be copied into another tab in the Watch pane, or into another pane or window.

Timed Update when Running

If you are using RealMonitor or an RTOS extension, this 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 option is enabled when:

- RealMonitor is activated (see the chapter that describes configuring custom targets in *RealView Debugger v1.8 Target Configuration Guide* for details)
- you are in RSD mode and where supported by the underlying debug target (see the chapter that describes RTOS support in *RealView Debugger v1.8 Extensions User Guide* for details).

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.

Update Window

Updates the contents of the Watch pane. Use this when **Timed Update when Running** and **Automatic Update** have been disabled. Select this to update the current tab.

Enter New Expression...

Displays a prompt box where you can enter the expression to be watched. This displays the name and current value in the current tab. Click the required tab before selecting this option.

Automatic Update

Refreshes the Watch pane as soon as execution stops. This is enabled by default.

Recompute Expression with same Context

If you watch an expression, the result is evaluated based on the current context. Select this option to recompute expressions using the context when set.

Show char* and char[] as strings

Displays values as strings where appropriate. This is enabled by default.

Show integers in hex

Displays all integers in hexadecimal format. Disabling this option displays all integers in decimal. This is enabled by default.

Properties... Displays a text description of the item under the cursor.

Viewing watches

As you set watches, expressions are added to the Watch pane, shown in Figure 6-15.

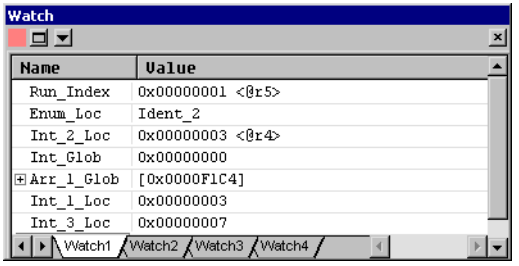

Figure 6-15 Watch pane with watches

The entries correspond to the watches you set, in the order that you set them. Each expression is shown giving the `Name` and `Value`. You can expand the column headings by dragging on the boundary marker to make the details easier to read.

To delete an expression, highlight it and press Delete. There is no undo for this operation but saving the watches in your personal Favorites List enables you to reinstate any deleted entry.

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. The Watch pane enables you to carry out these operations and gives access to context menus where editing options are available.

Figure 6-16 shows an example Watch pane, with several expressions already set.

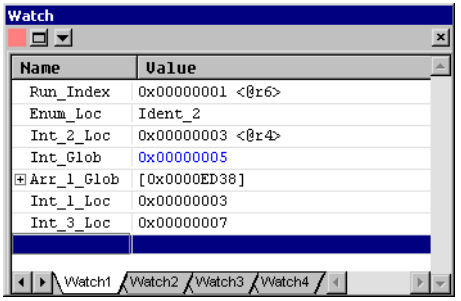

Figure 6-16 Watches in the Watch pane

One watched value is an array, shown by the plus sign appended to the variable name. Click on the plus sign to expand the view and display the array elements.

Note

If the chosen array is very large, RealView Debugger warns you before expanding the view. Click either **Yes** to expand the array, or **No** to cancel the operation.

You can access context menus, inside the Watch pane, to control watch options and edit watches directly, see:

- *Using the Name menu*
- *Using the Value menu* on page 6-44
- *Using the pane context menu* on page 6-45
- *Editing watches* on page 6-46.

Using the Name menu

If you have set up expressions, right-click on a chosen entry `Name` to display the **Name** menu. This context menu provides options that operate on selected entries only:

Update Updates the displayed value for the chosen expression. This is applied in combination with any update options you set from the **Pane** menu.

Format... Displays a selection box to specify the format for the watched expression.

Highlight the required format from the list of available formats. You can also cast top level expressions, including casting to array, for example `char*` and `char[12]`.

Click **OK** to confirm your choice. This closes the selection box and the new format is applied to the chosen expression.

If a variable contains multibyte characters, you can choose to view it using `ASCII`, `UTF-8`, or `Locale` encoding.

Break at... If enabled, select this option to set a breakpoint at this location.

BreakIf... If enabled, select this option to set a conditional breakpoint.

Add from Favorites...

Displays the Favorites Chooser/Editor dialog box where data values saved in your personal favorites can be added to the Watch pane (see *Using the Favorites Chooser/Editor dialog box* on page 8-29 and *Saving watches as favorites* on page 6-46 for details).

View Memory at Value

Select this option to use the chosen value as the starting address for a memory display.

View Memory at Address

Select this option to use the address of the chosen item as the starting address for a memory display.

Properties... Displays a text description of the item under the cursor.

Note

The first time you perform the **View Memory at Value** or **View Memory at Address** operations in a debugging session, a new instance of the Memory pane is displayed as a floating pane. Subsequent view memory operations use this instance of the Memory pane to display the memory contents, even if you have docked that pane.

If you change the docked instance of the Memory pane to a different pane view (such as Process Control), and you later perform the **View Memory At Value** operation, the docked Memory pane is redisplayed to the right of the changed pane view (Process Control in this example).

This Memory pane is also used if you perform the view memory operations from the Register pane or the File Editor pane.

Using the Value menu

With a watch set, you can right-click on a chosen entry `Value` to display the **Value** menu. This context menu provides options that operate on selected watches only:

Update Updates the displayed value for the chosen expression. This is applied in combination with any update options you set from the **Pane** menu.

Format... Displays a selection box where you can highlight the required format for the expression from the list of available formats.

If a variable contains multibyte characters, you can choose to view it using `ASCII`, `UTF-8`, or `Locale` encoding.

Set to 0 Sets the value of the chosen expression to zero, where allowed. An error message is displayed if you try to set a value to zero when not permitted.

Increment Adds 1 to the value of the chosen expression. An error message is displayed if you try to increment a value when not permitted.

Decrement Subtracts 1 from the value of the chosen expression. An error message is displayed if you try to decrement a value when not permitted.

Set from Favorites...

Displays the Favorites Chooser/Editor dialog box where data values saved in your personal Favorites List can be inserted into the specified location (see *Using the Favorites Chooser/Editor dialog box* on page 8-29 and *Saving watches as favorites* on page 6-46 for details).

Recent expressions

The rest of this menu contains a list of recently-used variables and data values. You can re-use entries from this list as required.

Using the pane context menu

If you right-click anywhere inside an empty entry in the Watch pane, a short context menu is displayed. This provides the options:

Update All Updates the details for all the expressions currently displayed in the Watch pane.

Add from Favorites...

Displays the Favorites Chooser/Editor dialog box where expressions saved in your personal Favorites List can be added to the Watch pane (see *Using the Favorites Chooser/Editor dialog box* on page 8-29 and *Saving watches as favorites* on page 6-46 for details).

Editing watches

You can use in-place editing to change expressions in the Watch pane, and to add new ones:

1. Double-click in the name you want to change, or press Enter if the item is already selected. The name is enclosed in a box with the characters highlighted to show they are selected (pending deletion).
2. Enter the new name, or move the cursor to change the existing expression, or add a cast.
3. Press Enter to store the new name.

If you press Escape then any changes you made in the highlighted field are ignored.

6.5.4 Saving watches as favorites

You can create watches and then add them directly to your personal Favorites List or you can add watches that you have been using in the current debugging session. This section explains the steps to follow to do both.

Creating a watch favorite

To create a Watch favorite, right-click inside a blank entry of the Watch pane to display the **Name** menu, and then select **Add from Favorites...**. This displays the Favorites Chooser/Editor dialog box. If this is the first time you have used watches in RealView Debugger, the display list is empty.

To create a new watch and add it to your Favorites List:

1. Click **New** to display the New/Edit Favorite dialog box shown in Figure 6-17.

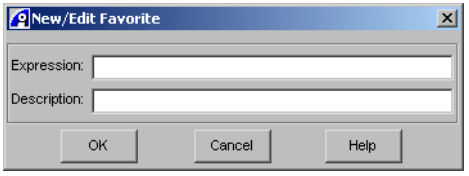

Figure 6-17 New/Edit Favorite dialog box

2. Enter the expression to be watched, for example `Ptr_Comp`.
3. Enter a short text description to help you to identify the watch for future use, for example `my watch favorite`.

   This is optional.

4. Click **OK** to confirm the entries and close the New/Edit Favorite dialog box.

   The Favorites Chooser/Editor dialog box is displayed with the newly-created watch shown in the display list (shown in Figure 6-18). See *Using the Favorites Chooser/Editor dialog box* on page 8-29 for details.

   Duplicate entries are not permitted in the Favorites List.

Saving existing watches as favorites

With several watches already set, RealView Debugger lets you choose which to add to your Favorites List so that they are available for re-use in future debugging sessions or with other target configurations of your application program.

To add existing watches to your Favorites List:

1. Highlight an expression in the Watch pane that you want to add to your Favorites List.

2. Right-click on the `Name` and select **Add from Favorites...** from the **Name** menu. This displays the Favorites Chooser/Editor, shown in Figure 6-18.

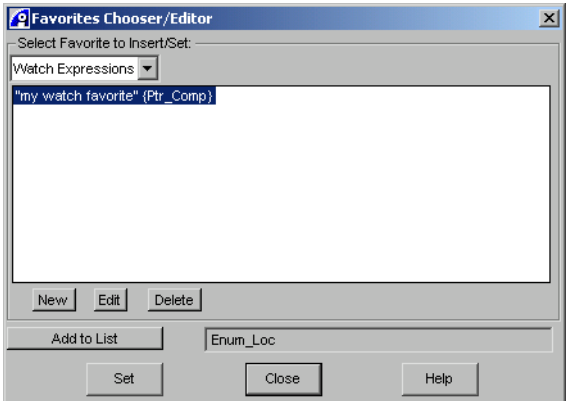

Figure 6-18 Existing watches in the Favorites Chooser/Editor

   The display list shows any watches already saved in your Favorites List. The data field now shows the chosen expression.

3. Click **Add to List** to add the specified expression to your Favorites List. This displays the New/Edit Favorite dialog box shown in Figure 6-19 on page 6-48.

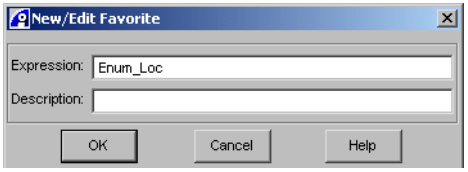

Figure 6-19 Adding a new favorite

The Expression field contains the chosen watch and you can enter a short text description to help you identify the watch favorite. This is optional.

4. Click **OK** to confirm the watch details and close the dialog box.

The Favorites Chooser/Editor dialog box is displayed showing the new watch in the display list. Because this watch is already set, click **Close** to close the dialog box. If required, set another watch from your Favorites List before closing the dialog box.

The edited Favorites List is saved to your `exphist.sav` file when you close RealView Debugger.

Saving data values as favorites

With several watches already set, you can right-click on the `Value` for a specified expression and display the **Value** menu. This context menu includes the option **Set from Favorites...** to specify a data value to be set.

Select this option to display the Favorites Chooser/Editor dialog box where you can:

- save an existing data value as an entry in your Favorites List so that it can be re-used later in this debugging session
- take a data value already saved and use it to set the starting value for the specified watch.

Note

If you are using RealView Debugger on non-Windows platforms, the history file is only created if you create and save a favorite, for example a breakpoint or watchpoint. See Appendix B *RealView Debugger on Sun Solaris and Red Hat Linux* for details.

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:

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 7-4
- *Working with Flash* on page 7-5.

The last section in this chapter contains examples using interactive operations see:

- *Examples of interactive operations* on page 7-10.

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.

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*.

7.3.1 From the Memory pane

If you are working in the Memory pane, you can access memory operations:

1. Select **View** → **Memory** to display the Memory pane.
2. Right-click on a memory cell, or byte, that is black or green, that is where the type is ROM, Flash, or modifiable, to display the **Memory** context menu.
3. Select the required option, for example **Set Memory Interactive...** to display the Interactive Memory Setting dialog box (see *Setting memory* on page 7-10 for instructions on how to use this 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** → **Stack** to display the Stack pane.
2. Right-click on a memory cell, or byte, to display the **Memory** context menu.
3. Select the required option, for example **Set Memory Interactive...** to display the Interactive Memory Setting dialog box (see *Setting memory* on page 7-10 for instructions on how to use this dialog box).

7.4 Working with Flash

To use RealView Debugger to control Flash memory on your chosen debug target, you must:

- have access to an appropriate *Flash MEthod* (FME) file
- have access to a *Board/Chip Definition* (BCD) file that specifies the memory map of your debug target, and the FME file to use
- assign the BCD file to your debug target.

Depending on your current target, this might mean that you must first:

- create the FME file for your Flash type
- create and set up the BCD file.

See the chapter that describes Flash programming with RealView Debugger in the *RealView Debugger v1.8 Target Configuration Guide* for detailed instructions on creating an FME file, and creating, setting up and assigning a BCD file to your debug target.

When you have configured your debug target to use the required BCD and FME files, you can program your Flash image into the Flash device. This section describes how to program a Flash image into the Flash device on your debug target. It includes:

- *Flash Method files*
- *Flash examples* on page 7-6
- *Flash programming* on page 7-6
- *Using the Flash Memory Control* on page 7-8.

7.4.1 Flash Method files

FME files include code to:

- enable you to write to the Flash on your debug target
- perform read, write, and erase operations
- describe the way the Flash is configured on the bus.

Example files are included for all supported Flash devices as part of the root installation.

If you have a custom board or custom Flash type that is not one of those supported by RealView Debugger, you must create your own FME file. See the chapter that describes Flash programming with RealView Debugger in the *RealView Debugger v1.8 Target Configuration Guide* for detailed instructions on how to create an FME file.

7.4.2 Flash examples

The root installation contains a directory of examples for supported Flash devices. These examples contain the prebuilt FME files that are required to program the supported Flash devices.

All the Flash examples are located in:

install_directory\RVD\Core\...\flash\examples

Files are collected in subdirectories based on the target board or Flash device.

7.4.3 Flash programming

Before you can use RealView Debugger to control a Flash device on your target, you must:

- describe the Flash memory chip in the memory map
- ensure that you have a correctly configured FME file.

The following example describes how to program the Flash memory on an ARM® Integrator™/AP board that is connected using RealView ICE. It assumes that your debug target is correctly configured to use the BCD file and FME file for the Integrator/AP board. The Integrator/AP FME file is installed as part of the root installation, in the Flash examples directory ...\IntegratorAP (see *Flash examples*).

Note

If you program the Flash on an ARM Integrator board using this release of RealView Debugger, you bypass the *ARM Firmware Suite* (AFS) Flash library system information blocks. These blocks are used by the AFS Flash Library, and are stored at the end of each image written to Flash. If you rely on these blocks to keep track of what is in the Flash memory of your target, keep a record of the state and recreate it after working through the example.

Programming an image into Flash

To program an image, you ask RealView Debugger to write to the Flash memory region that you have defined in the board file. The Integrator/AP Flash starts at memory address 0x24000000.

The following procedure uses the Load File to Target dialog box to load a Flash image in preparation for writing to Flash. However, you can also program the image into Flash using the Upload/Download file to/from Memory dialog box. For instructions on how to do this, see *Loading Flash memory with the Upload/Download file from/to Memory dialog box* on page 7-18.

To write an image to the Integrator/AP Flash:

1. Build an image compiled to run with code at `0x24000000` and that has data in RAM.

   This example uses the `dhrystone` program stored in:

   `install_directory``\RVDS\Examples\...\dhrystone`

   To rebuild the `dhrystone` image with modified linker options:

   a. Open the `dhrystone` project.

   b. Select **Project → Project Properties** to open the **Project Properties** window.

   c. Click on the entry `*BUILD` in the left hand pane and double-click on the `Link_Advanced` entry.

   d. Right-click on the `Ro_base` entry in the right-hand pane and select **Edit Value** from the context menu. Enter the value `0x24000000`.

   e. Right-click on the `Rw_base` entry in the right-hand pane and select **Edit Value** from the context menu. Enter the value `0x8000`.

   f. Select **Save and Close** in the Project Properties window.

      Wait until RealView Debugger has finished generating the makefiles.

   g. Select **Build → Rebuild All** to build the project.

2. Connect to the target device (Integrator/AP in this example).

3. Click on the blue hyperlink in the File Editor pane to load the `dhrystone.axf` image.

   This displays the Flash Memory Control dialog box (see Figure 7-1 on page 7-8). RealView Debugger queues the image in preparation for writing to Flash.

4. Click **Write** to program the image into Flash.

   Note

   Wait for the write to complete before continuing.

   See *Using the Flash Memory Control* on page 7-8 for more details on how to use the Flash Memory Control dialog box.

5. Click **Close** to close the Flash Memory Control dialog box.

6. Click on the **Cmd** tab in the Output pane to see the Flash operations.

7. Select **View → Memory Map tab** to display the **Map** tab where you can see the Flash memory area (green icon) on the Integrator/AP board. Expand the Flash memory map entry to see the image and Flash details.

7.4.4 Using the Flash Memory Control

The Flash Memory Control dialog box shown in Figure 7-1 enables you to perform various operations on a Flash device.

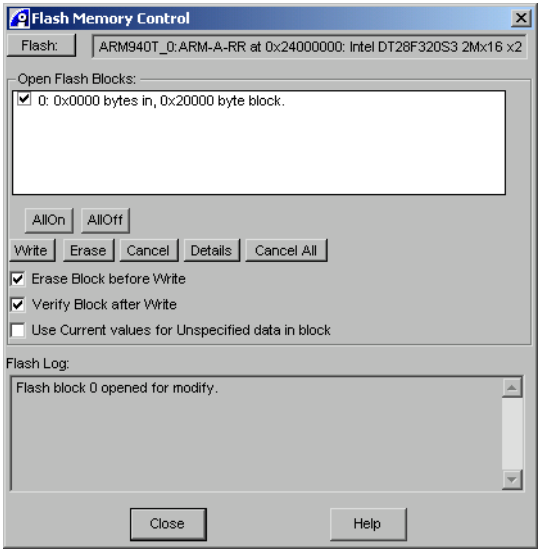

Figure 7-1 Flash Memory Control dialog box

The Flash Memory Control dialog box consists of a display list, a read-only data field, a Flash Log, and control buttons:

Flash: Click this button to view details about the Flash. The data field next to the **Flash** button describes the connection, Flash start address, and type of Flash being used.

Open Flash Blocks:

Lists the Flash blocks that are currently open for access. A check box is associated with each block. When this check box is checked, any Flash operations are performed on that block. When you first write an image to Flash, only those Flash blocks that are affected by the image are selected.

AllOn Selects all entries in the Open Flash Blocks list as indicated by a check in the accompanying check box. This enables you to carry out operations on all the open blocks.

AllOff Unselects all entries in the Open Flash Blocks list as indicated by no check in the accompanying check box.

Write Writes data to the specified blocks of Flash.

Erase Erases every specified block of Flash. This normally sets every byte to `0x00` or `0xFF` depending on the type of Flash being used.

Cancel Abandons any changes made to the specified blocks of Flash.

Cancel All Abandons all changes to the Flash contents.

Details Displays an information box describing the type of Flash being used.

Erase Block before Write

Select this check box to erase the Flash block before performing the write operation.

Verify Block after Write

Select this check box to verify the Flash block, against the data source, after performing the write operation.

Use Current values for Unspecified data in block

Specifies that the original contents are to be maintained unless modified by the current operation. If unselected, the erase values are used.

Only use this option if you are updating part of the Flash block and you want to retain the current values in the rest of the block. This check box is unselected by default. If you select this option, RealView Debugger reads and then writes the entire block. This might take some time to complete.

Flash Log: Displays a log of operations carried out on the selected Flash blocks.

Close Closes the Flash Memory Control dialog box.

Help Displays online help for this dialog box.

See *Setting Flash memory* on page 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 7-12
- *Downloading memory to a file* on page 7-14
- *Comparing memory with file contents* on page 7-16
- *Filling memory with a pattern* on page 7-16
- *Loading Flash memory with the Upload/Download file from/to Memory dialog box* 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. Select **Edit → Advanced → Show Line Numbers** to display line numbers.

   This is not necessary but might help you to follow the examples.
3. Click on the **Src** tab to view the source file `dhry_1.c`.
4. Set a default breakpoint by double-clicking on line 301.
5. Click **Go** to start execution.
6. Enter `5000` when asked for the number of runs.

   The program starts and then stops when execution reaches the breakpoint at line 301. The red box marks the location of the PC when execution stops.
7. Select **View → Memory** to display the Memory pane.

   Start addresses can be set using in-place editing or using the **Memory** context menu.
8. Right-click in the first address in the window to display the **Memory** context menu.
9. Select **Set Start Address...** and enter `0x000088A0` as the new start address.
10. Highlight the first byte at this address. In this example, this is `0x08`.
11. Right-click and select **Set Memory Interactive...** from the context menu to display the Interactive Memory Setting dialog box, shown in Figure 7-2.

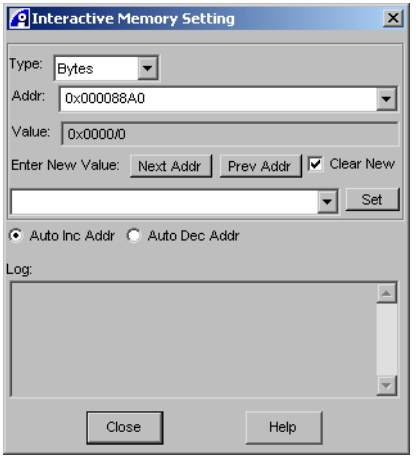

Figure 7-2 Interactive Memory Setting dialog box

12. Enter the required memory settings:

    Type: Select the display format. See *Formatting options* on page 6-21 for details of the memory formats.

    Addr: The address where the memory setting starts. Depending on the method used to display this dialog box, this field is already populated, as in this example.

    The address must be entered in hexadecimal format, for example `0x000088A0`.

    Value: This read-only data field shows the current value, in hexadecimal and decimal formats, at the specified memory location.

    Enter New Value:

    Enter the value to be set at the current location, for example `0x08` or `8` (decimal).

    If the Memory pane is configured to update automatically, clicking **Set** immediately updates the memory contents. This is the default setting in the **Pane** menu.

    If you press Enter with no value in the Value data field, RealView Debugger moves automatically to the next, or previous, location.

    Next Addr

    Moves the target address to the next location by adding 1 to the address displayed in the Addr data field. This depends on the size of the current type.

Prev Addr

Moves the target address to a new location by subtracting 1 from the address displayed in the Addr data field. This depends on the size of the current type.

Clear New

Automatically clears any value entered in the Enter New Value data field ready to accept another value. By default, this feature is enabled.

Auto Inc Addr

If selected, this radio button instructs RealView Debugger to increment the target address automatically ready to accept a new setting.

Auto Dec Addr

If selected, this radio button instructs RealView Debugger to decrement the target address automatically ready to accept a new setting.

Log: Displays a log of the changes you have made. This log is shown when you next display the dialog box.

13. See the log updated to show your change.
14. Click **Close** to close the Interactive Memory Setting dialog box.
15. Double-click on the red marker disc to clear the breakpoint at line 301.

Changed values are displayed in the Memory pane in the usual way. That is, updated values are displayed in dark blue or light blue, depending on when they last changed.

7.5.2 Setting registers

To set register contents:

1. Select **Target → Reload Image to Target** to reload the image `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Set a default breakpoint by double-clicking on line 301.
4. Click **Run** to start execution.
5. Enter `5000` when asked for the number of runs.

   The program starts and then stops when execution reaches the breakpoint at line 301. The red box marks the location of the PC when execution stops.
6. Select **View → Registers** to display the Register pane.

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

   This dialog contains almost the same controls as the Interactive Memory Setting dialog box described in *Setting memory* on page 7-10.

8. Set up the required register settings:

   Register: Enter the register to change, for example `@R4`. Press Enter to confirm your choice.

   If required, use the drop-down arrow to select a previously used register from the stored list.

   Value: This read-only data field shows the current value, in hexadecimal and decimal formats, for the specified register.

   Enter New Value:

   Enter the value to be set, in hex or decimal, for example `0xCC4`.

   All changed registers are displayed in blue.

   Next Addr

   Moves to higher register by adding 1 to the register number displayed in the Register field.

   Prev Addr

   Moves to a lower register by subtracting 1 from the register number displayed in the Register field.

   Clear New

   Automatically clears any value entered in the Enter New Value data field ready to accept another value. By default, this feature is enabled.

   Auto Inc Reg

   If selected, this radio button instructs RealView Debugger to increment the register number automatically ready to accept a new setting.

   Auto Dec Addr

   If selected, this radio button instructs RealView Debugger to decrement the register number automatically ready to accept a new setting.

   Log: Displays a log of the changes you have made. This log is shown when you next display the dialog box.

9. See the log updated to show your change.

10. Click **Close** to close the Interactive Register Setting dialog box.

11. Double-click on the red marker disc to clear the breakpoint at line 301.

7.5.3 Downloading memory to a file

To download a memory range into a file:

1. Select **Target → Reload Image to Target** to reload the image `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Set a default breakpoint by double-clicking on line 301.
4. Click **Go** to start execution.
5. Enter `5000` when asked for the number of runs.

   The program starts and then stops when execution reaches the breakpoint at line 301. The red box marks the location of the PC when execution stops.
6. Select **Debug → Memory/Register Operations → Upload/Download Memory file...** from the Code window main menu to display the Upload/Download file from/to Memory dialog box, shown in Figure 7-3.

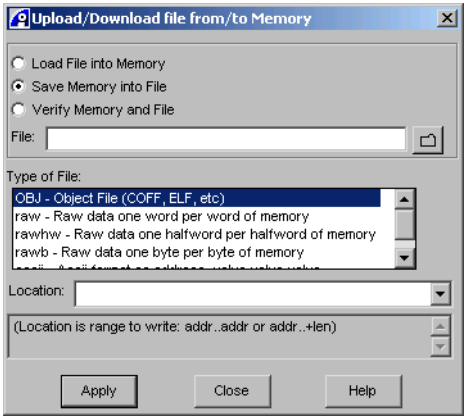

Figure 7-3 Upload/Download file from/to Memory dialog box

7. Specify the operation and set up the controls, as follows:
   a. Select the **Save Memory into File** radio button. This instructs RealView Debugger to access the specified memory block, read the contents, and write them to the given file.
   b. In the **File** text box, enter the full pathname of the file to use to read/write memory values.

c. In the **Type of File** section of the dialog, select the data type to be used in the specified file where:
   - `OBJ` specifies an object file in the standard executable target format, for example ARM-ELF for ARM architecture-based targets
   - `raw` specifies a data file as a stream of 32-bit values
   - `rawhw` specifies a data file as a stream of 16-bit values
   - `rawb` specifies a data file as a stream of 8-bit values
   - `ascii` specifies a space-separated file of hexadecimal values.

d. Define the start location of the memory block.

   When writing memory, specify a range as an address range or as a start address and length, for example:
   - `0x88A0..0x8980`
   - `0x88A0..+1000`
   - `main..+1000`

   If required, use the drop-down arrow to select a previously used location from the stored list.

   Note

   If you are reading from a file to memory, you must specify a start location. The range can be left blank, unless the data type is binary.

   If you are writing to a file from memory, you must specify a start location and a range.

8. Click **Apply** to create and write the specified file. If the specified file already exists, you are prompted to overwrite it.
9. Click **Close** to close the Upload/Download file to/from Memory dialog box.
10. Double-click on the red marker disc to clear the breakpoint at line 301.

Note

If you are writing memory to a file and the specified file already exists, RealView Debugger warns of this and asks for confirmation before overwriting the file contents.

RealView Debugger warns you if the memory transfer is going to take a long time to complete. When reading or writing memory contents, you must be aware that:

- There is no limit on the size of file that RealView Debugger can handle.
- The time taken to complete the operation depends on the access speeds of your debug target interface.

7.5.4 Comparing memory with file contents

To verify a file:

1. Ensure that you have downloaded a memory range into a file as described in *Downloading memory to a file* on page 7-14.
2. Select **Debug → Memory/Register Operations → Upload/Download Memory file...** from the Code window main menu to display the Upload/Download file from/to Memory dialog box (see Figure 7-3 on page 7-14).
3. Select **Verify Memory and File**.
4. Specify the file to be compared.
5. Specify the start address to be compared. This must be within the range specified in the memory file.
6. Click **Apply** to compare the file contents with the specified memory block.
7. Click on the **Cmd** tab in the Output pane and view the results, for example:

   ```
   verifyfile,ascii,gui "D:\ARM\RVD\Test_files\memory_file_3"
   Mismatch at Address 0x000088B6: 0x8E vs 0x8F
   ```

   The first mismatch is identified and the location reported. Any mismatches after this location are not reported.
8. Click **Close** to close the Upload/Download file from/to Memory dialog box.
9. Double-click on the red marker disc to clear the breakpoint at line 301.

7.5.5 Filling memory with a pattern

To fill a specified area of memory with a predefined pattern:

1. Select **Target → Reload Image to Target** to reload the image `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Set a default breakpoint by double-clicking on line 301.
4. Click **Run** to start execution.
5. Enter `5000` when asked for the number of runs.

   The program starts and then stops when execution reaches the breakpoint at line 301. The red box marks the location of the PC when execution stops.
6. Select **View → Memory** to display the Memory pane.

7. Double-click in the address column of the Memory pane, and enter `0x88A0` to view the memory contents at that location.

8. Select **Debug → Memory/Register Operations → Fill Memory with Pattern...** from the Code window main menu to display the Fill Memory with Pattern dialog box, shown in Figure 7-4.

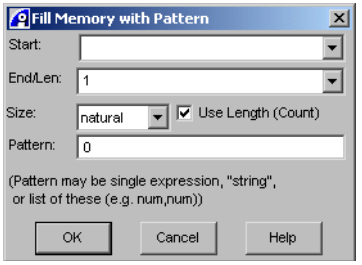

Figure 7-4 Fill Memory with Pattern dialog box

9. Set up the required memory settings:

   Start: Enter the start address for the memory range to be filled, for example `0x88A0`.

   If required, use the drop-down arrow to select a previously used start address from the stored list.

   End/Len: By default, the memory area that is filled is defined by a start address and a length. Enter the length of the memory block to be filled in this data field, for example `14` (decimal).

   The specified length must be given relative to the data type given in the **Size** data field.

   If the **Use Length (Count)** check box is unselected, you can specify the address that marks the end of the memory block.

   Size: Enter the data type to be used in the file where:

   - `natural` indicates the format specified as native for the debug target
   - `byte` indicates support for 8-bit signed and unsigned byte form
   - `half-word` indicates support for 16-bit signed and unsigned halfwords
   - `long-word` indicates support for 32-bit signed and unsigned words.

Use Length (Count)

By default, the memory block to be filled is defined by a start address and the length. If this check box is unselected you can specify an address to mark the end of the filled block.

Pattern: Enter the pattern to be used as the fill, for example:

```
"AB"
0,1,0,1,0
```

Note

Strings must be enclosed in quotes.

10. Click **OK** to confirm your settings and close the Fill Memory with Pattern dialog box. Memory contents are rewritten and the Memory pane is automatically updated with changed values displayed in blue.

11. Double-click on the red marker disc to clear the breakpoint at line 301.

When filling memory blocks, you must be aware of the following:

- All expressions in an expression string are padded or truncated to the size specified by the Size value if they do not fit the specified size evenly.
- If the number of values in an expression string is less than the number of bytes in the specified address range, RealView Debugger repeats the pattern and so might fill an area in excess of the specified block, for example specify a pattern of 10 bytes and a fill area of 16 bytes. RealView Debugger repeats the pattern twice and so fills a block of 20 bytes.
- If more values are given than can be contained in the specified address range, excess values are ignored.
- If a pattern is not specified, RealView Debugger displays an error message.

7.5.6 Loading Flash memory with the Upload/Download file from/to Memory dialog box

This example assumes that you have built the `dhrystone.axf` image as described in *Programming an image into Flash* on page 7-6, step 1, and that you are using the Integrator/AP board as the target.

To load Flash memory:

1. Connect to the Integrator/AP target.

2. Select **Debug → Memory/Register Operations → Upload/Download Memory file...** from the Code window main menu to display the Upload/Download file from/to Memory dialog box, shown in Figure 7-3 on page 7-14.

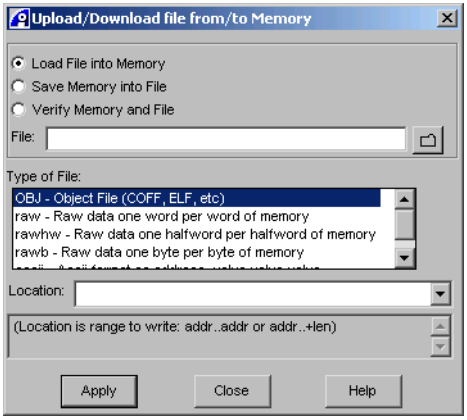

Figure 7-5 Upload/Download file from/to Memory dialog box

3. Specify the operation and set up the controls, as follows:
   a. Select the **Load File into Memory** radio button. This instructs RealView Debugger to load the chosen file starting at the specified address.
   b. In the **File** text box, enter the full pathname of the `dhrystone.axf` you built in step 1.
   c. In the **Type of File** section of the dialog, select `OBJ`. This specifies an object file in the standard executable target format, for example ARM-ELF for ARM architecture-based targets.
   d. Define the start location of the Flash memory.

      For the Integrator/AP board, enter `0x24000000`.

4. Click **Apply**.

   RealView Debugger queues the image in preparation for writing to Flash, and displays the Flash Memory Control dialog box (see *Using the Flash Memory Control* on page 7-8).

   Note

   Although RealView Debugger updates the Memory pane and the File Editor pane with the image information, the image is yet not written to the Flash.

5. Click **Write** to write the image to Flash.

Note

Wait for the write to complete before continuing.

6. Click **Close** to close the Flash Memory Control dialog box.
7. Click **Close** to close the Upload/Download file to/from Memory dialog box.

7.5.7 Setting Flash memory

Flash memory blocks open for access when you write to Flash, for example when you load an image. This displays the Flash Memory Control dialog box (see *Using the Flash Memory Control* on page 7-8 for details).

To write to Flash memory interactively:

1. Connect to your target, load an image, and write to Flash, as described in *Working with Flash* on page 7-5.
2. Select **View** → **Memory** to display the Memory pane.

   Start addresses can be set using in-place editing or using the **Memory** context menu.
3. Right-click in the first address in the window to display the **Memory** context menu.
4. Select **Set Start Address...** and enter `0x24000000` as the new start address.

   This is colored green, indicating Flash.
5. Right-click in the first byte at this address.
6. Select **Set Value...** from the context menu.
7. Enter the new value `0xA0` at the prompt.
8. Click **Set** to confirm this value. This displays the Flash Memory Control dialog box to enable you to access the open Flash, shown in Figure 7-1 on page 7-8.
9. If you want to view details of the Flash memory, click **Flash** in the Flash Memory Control dialog box. This displays details of the Flash memory, as shown in Figure 7-6 on page 7-21.

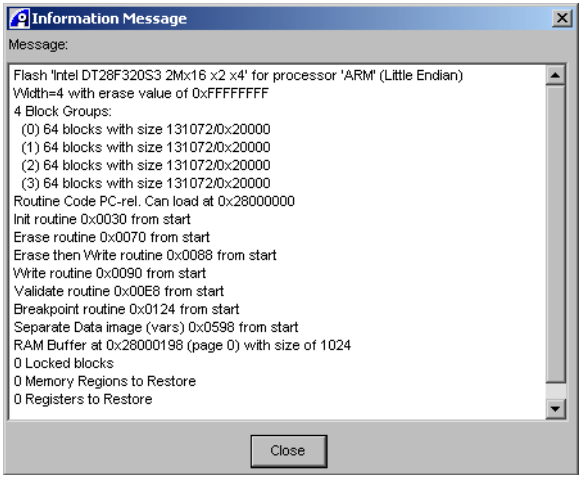

Figure 7-6 Flash memory details

You can also click **Details** to display information about the open Flash block.

Click **Close** to close the information box.

10. In the Flash Memory Control dialog box, ensure that the **Erase Block before Write** option is checked.

11. Click **Write** to write to the chosen Flash location. Monitor the changes in the Memory pane as memory is updated. The Flash Log confirms the Flash operation.

12. Click **Close** to close the Flash Memory Control dialog box.

Note

You can also select **Debug → Memory/Register Operations → Flash Memory Control...**, from the Code window main menu, to display the Flash Memory Control dialog box during your debugging session.

Chapter 8
Working with Browsers and Favorites

RealView® Debugger provides browser panes and list dialog boxes to help with debugging tasks and monitor your program during execution. This chapter describes how to access these panes and dialog boxes from the Code window, and how to use them. It contains the following sections:

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.

This section includes:

- *Scope of symbols in RealView Debugger*
- *Browser panes*
- *List browser dialog boxes*.

8.1.1 Scope of symbols in RealView Debugger

RealView Debugger uses scope to determine the value of a symbol. Any symbol value available to a C or C++ program at the current PC is also available to RealView Debugger.

Variables can have values that are relevant within:

- a specific class only, that is *class scope*
- a specific function only, that is *local scope*
- a specific file only, that is *static global scope*
- the entire process, that is *global scope*.

For full details on scope and scoping rules see the chapter that describes working with the CLI in *RealView Debugger v1.8 Command Line Reference Guide*.

8.1.2 Browser panes

The following browser panes are available:

- Data Navigator, to quickly locate a module, function, or variable (see *Using the Data Navigator pane* on page 8-5)
- Symbol Browser, for viewing C++ classes (see *Using the Symbol Browser pane* on page 8-17).

8.1.3 List browser dialog boxes

You can access list browser dialog boxes from various routes when working with RealView Debugger. These browsers enable you to select specific locations when you are performing operations such as:

- setting breakpoints
- setting watchpoints to monitor the contents of the specified location
- viewing the memory from the specified location.

To view a list browser:

1. Display the required pane, if the required pane is not in view:
   - **View → Break/Tracepoints** to display the Break/Tracepoints pane
   - **View → Watch** to display the Watch pane
   - **View → Memory** to display the Memory pane.

2. Click on the **Pane** menu button in the pane toolbar.
3. Select the required option:
   - On the Break/Tracepoints pane menu, select **Set/Edit Breakpoint...** to display the Set Address/Data Breakpoint dialog box. Also, see *Accessing the list browsers from the breakpoint dialog boxes* on page 8-4.
   - On the Watch pane menu, select **Enter New Expression...** to display the expression prompt box.
   - On the Memory pane menu, select **Set Start Address...** to display the expression prompt box.

4. Click on the drop-down arrow to the right of the field to display the options list menu.
5. Select the required browser from the menu:

   <Function list...>

   Displays the Function List dialog box, described in *Using the Function List dialog box* on page 8-19.

   <Variable list...>

   Displays the Variable List dialog box, described in *Using the Variable List dialog box* on page 8-21.

   <Module list...>

   Displays the Module/File List dialog box, described in *Using the Module/File List dialog box* on page 8-23.

   <Register list...>

   Displays the Register List dialog box, described in *Using the Register List Selection dialog box* on page 8-25. This option is only available from the various dialog boxes used to set breakpoints, see *Accessing the list browsers from the breakpoint dialog boxes* on page 8-4.

Because the browsers are used only to make a selection, there are no controls for debugging operations such as **Watch**, **Break**, or **Show Type Info**.

Accessing the list browsers from the breakpoint dialog boxes

The list browsers are also accessible from the following breakpoint dialog boxes:

- Simple Break if X (see *Using the Simple Break if X dialog box* on page 4-21)
- Simple Break if X, N times (see *Using the Simple Break if X, N times dialog box* on page 4-35)
- Simple Break if X, when Y is True (see *Using the Simple Break if X, when Y is True dialog box* on page 4-37)
- HW Break if in Range (see *Using the HW Break if in Range dialog box* on page 4-24)
- HW While in function/range, Break if X (see *Using the HW While in function/range, Break if X dialog box* on page 4-26)
- HW Break if X, then if Y (see *Using the HW Break if X, then if Y dialog box* on page 4-27)
- HW Break on Data Value match (see *Using the HW Break on Data Value match dialog box* on page 4-29).

8.2 Using the Data Navigator pane

The Data Navigator pane enables you to:

- quickly locate a module, function, or variable in one or more images that are loaded on the debug target
- perform various operations, such as setting breakpoints on functions or variables.

This section describes how to use the Data Navigator pane, and includes:

8.2.1 Displaying the Data Navigator pane

To display the Data Navigator pane, shown in Figure 8-1, select **View → Data Navigator** from the Code window main menu.

Figure 8-1 Data Navigator pane

8.2.2 Data Navigator pane interface components

The Data Navigator contains the following interface components:

Filter field Use this to quickly locate a function. This field is updated as you choose an image, module, and function. Alternatively, if you know the module and function name in the image, you can enter this directly.

See *Applying a filter* for more details on how to specify filters.

Goto Variables check box

By default, when you double-click on a module entry in the **Modules** tab, the **Functions** tab is selected. Check this box to have the **Variables** tab selected instead.

Images tab Lists the images that are loaded on the current debug target. See *Working with the Images tab* on page 8-9 for details on using this tab.

Modules tab Lists the modules in all images, or in the chosen image. See *Working with the Modules tab* on page 8-10 for details on using this tab.

Functions tab

Lists the functions in all images or modules, or in the chosen image or module. See *Working with the Functions tab* on page 8-11 for details on using this tab.

Variables tab

Lists the variables in all images or modules, or in the chosen image or module. See *Working with the Variables tab* on page 8-14 for details on using this tab.

8.2.3 Applying a filter

Using a filter enables you to narrow down the search when displaying the list of images, modules, functions, or variables in the Data Navigator pane. When you specify a filter, the list of functions and variables is filtered accordingly. That is, only those functions and variables that match the filter are displayed.

Filter syntax

The syntax for specifying a filter is:

- For modules:

  `@image_name\module_name\*`

  The `\*` suffix is optional.

- For functions:

  `@image_name\module_name\function_name`

- For variables:

  `@image_name\module_name\variable_name`

Module names qualify symbolic references. See *Module naming conventions* on page 4-7 for details on using module names.

Filter metacharacters

Table 8-1 shows the metacharacters you can use to specify the filter rule or rules. When entering a filter, characters are case sensitive, for example the filter `*DHRY*` returns a list of three modules but `*dhry*` returns an empty list.

When you have completed the filter, press Enter and the list is refreshed. By repeatedly entering filters, and pressing Enter, you can refine the search to focus on selected modules, files, functions or variables.

Table 8-1 Filter metacharacters

Metacharacter	Description
`*`	This operator matches any character or number of characters, for example `*DHRY*` matches `MY_DHRYSTONE_H` but not `Dhrystone_H` or `MY_DHR`.
`?`	This operator matches any single character, for example `*DHRY_?`. matches `MY_DHRY_A` but not `MY_DHRY_AB` or `DHRY_B`.
`[…]`	*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 `*[HN]*` matches `DHRY_H` and `UNNAMED_1` but not `STDLIB`. An empty list (`[]`) returns no results.
`^`	This operator is used inside a list, to represent a NOT action, for example `*_[^2]*` matches `DHRY_1` but not `DHRY_2`.
`-`	*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 `*_[A-Z]*` matches `DHRY_H` but not `DHRY_1` whereas `*_[^A-Z]*` matches `UNNAMED_1` but not `DHRY_H`.
`-`	Used as the first character in a filter, this operator means do not match. For example, `*SAM* -*HOST*` means match all names containing the string `SAM` except those that contain the string `HOST`.

8.2.4 Refining the list of functions and variables

You can refine the list of functions and variables in the Data Navigator pane by using a filter (see *Applying a filter* on page 8-6). In addition, you can also use the following options on the **Functions** context menu (see *Functions context menu* on page 8-12) and the **Variables** context menu (see *Variables context menu* on page 8-15):

Show Publics

List global or public functions or variables with scope over all parts of the program.

Show Statics

List static functions or variables.

Shows Labels

List code labels with scope over entire functions.

Show Locals

Displays local variables with scope within a function. This option is available only on the **Variables** context menu.

Show Library Symbols

List library symbols depending on the state of the **Show Publics**, **Show Statics**, **Show Locals**, and **Show Labels** options.

8.2.5 Setting a default breakpoint on a function

In the Data Navigator pane, you can set a default breakpoint on a chosen function. To do this:

1. Locate the function in the **Functions** tab.
2. Right-click on the function entry.
3. Select **Break** to set a breakpoint.

This sets a default breakpoint on the function. RealView Debugger uses the `BREAKINSTRUCTION` command to set the breakpoint:

- If the function is in RAM, a software breakpoint is set.
- If the function is in Flash or ROM, a hardware breakpoint is set. A warning message might also be issued, depending on your debug target.

For more details on default breakpoints, see *Setting default breakpoints* on page 4-15.

8.2.6 Setting a non-default breakpoint on a function

If you want to use the Data Navigator pane set a non-default breakpoint on a function, that is you want to set a breakpoint that has qualifiers or actions, do the following:

1. Right-click on the function, and select one of the following options from the **Functions** context menu:
   - **Show in Disassembly**
   - **Show Source**
   - **Scope To**.
2. Set the required breakpoint as described in the following sections:
   - *Setting unconditional breakpoints explicitly* on page 4-21
   - *Setting hardware breakpoints explicitly* on page 4-24
   - *Setting conditional breakpoints* on page 4-34
   - *Using the Set Address/Data Breakpoint dialog box* on page 4-41.

8.2.7 Setting a hardware breakpoint on a variable

In the Data Navigator pane, you can set a hardware access, read, or write breakpoint on a chosen variable. To do this:

1. Locate the variable in the **Variables** tab.
2. Right-click on the variable entry to display the context menu.
3. Select **Break...** from the context menu.

This displays the Set Address/Data Breakpoint dialog box, with the Location field containing the reference to the selected variable. Only **HW Access**, **HW Read**, and **HW Write** breakpoint types are valid for variables. For more details on using this dialog box, see *Using the Set Address/Data Breakpoint dialog box* on page 4-41.

8.2.8 Working with the Images tab

The **Images** tab in the Data Navigator pane lists the images that you have loaded onto the current debug target.

Images tab operations

You can perform the following operations with the **Images** tab:

- Sort the image names in ascending or descending order. To do this, click on the **Images** column header.

- Narrow the search for a module, function, or variable to a specific image. To do this, double-click on the image name. The **Modules** tab becomes the current tab (see *Working with the Modules tab*). Also:
  - — the **Functions** tab lists those functions in the image that you have elected to show in that tab
  - — the **Variables** tab lists those variables in the image that you have elected to show in that tab.

8.2.9 Working with the Modules tab

The **Modules** tab in the Data Navigator pane lists the modules in all images by default, or in the image you selected from the **Images** tab (see *Working with the Images tab* on page 8-9). You can also limit the list of modules by specifying a filter (see *Applying a filter* on page 8-6).

Modules tab operations

You can perform the following operations with the **Modules** tab:

- Sort the modules in ascending or descending order of Module Name, Filename, or Image. To do this, click on the appropriate column header.
- Narrow the search for a function or variable to a specific module. To do this, double-click on the required module entry:
  - — By default, the **Functions** tab becomes the current tab (see *Working with the Functions tab* on page 8-11). This tab lists those functions in the module that you have elected to show in that tab.
  - — If you have checked the **Goto Variables** check box, the **Variables** tab becomes the current tab (see *Working with the Variables tab* on page 8-14). This tab lists those variables in the module that you have elected to show in that tab.
- Locate the lowest address in memory that is occupied by the selected module, see *Modules context menu* on page 8-11 for details.

Modules context menu

To display the **Modules** context menu, right-click on a module entry. This menu has a single option:

Scope To Locates the lowest address in memory that is occupied by the chosen module in the current view of the File Editor pane. The address is that of the function with the lowest address in the module:

- If the **Dsm** tab is selected, the function is displayed in the disassembly view.
- If a source file tab is selected, for example `dhry_1.c`, and the chosen module matches that source file, the function is displayed.

  If the chosen module matches a source file that is not selected, RealView Debugger selects the source file. If the source file is not already open, then RealView Debugger opens it.

If the selected module is in an ARM library, then RealView Debugger is unable to display the source, and displays the following message in the **Src** tab:

`No source for context: module\function`

module`\`*function* refers to the function in the module that has the lowest address in memory, for example, `DIVISION_S\__rt_sdiv`, or `<Unknown>` if RealView Debugger is unable to determine the context.

In all cases where a module context can be determined, RealView Debugger also locates the module in the **Dsm** tab. Select the **Dsm** tab to view the start address for the module.

If RealView Debugger is unable to determine the context, then address `0x00000000` is chosen. For example, if you scope to a module that relates to a `.h` file, such as `DHRY_H`.

8.2.10 Working with the Functions tab

The **Functions** tab in the Data Navigator pane lists the functions in all modules by default, or in the module you selected from the **Modules** tab (see *Working with the Modules tab* on page 8-10). You can also limit the list of functions displayed by specifying a filter (see *Applying a filter* on page 8-6).

Functions tab operations

You can perform the following operations with the **Functions** tab:

- Sort the functions in ascending or descending order of Function Name, Address, Scope, Module, or Image. To do this, click on the appropriate column header.

- Various operations are available from the **Functions** context menu. To display this context menu, right-click on a function entry. See *Functions context menu* for a description of these operations.

Functions context menu

The **Functions** context menu contains the following options:

Show Publics

When enabled (checked), lists all global or public functions with scope over all parts of the image, or selected module.

Show Statics When enabled (checked), lists all static functions in the image, or selected module.

Show Labels When enabled (checked), lists all code labels with scope over the entire image, or selected module.

Show Library Symbols

When enabled (checked), lists the symbols in the libraries used by the image. The library symbols listed depends on the state of the **Show Publics**, **Show Statics**, and **Show Labels** options.

When disabled (unchecked), most library symbols are filtered out. This is the default.

Show in Disassembly

Locates the function in the disassembly view in the **Dsm** tab of the File Editor pane. If the disassembly view is not visible, RealView Debugger switches the code view to the **Dsm** tab.

You can also locate the function in the disassembly view by double-clicking on the function entry.

Show Source

Locates the function in your source code of the File Editor pane. If the source file is not open, RealView Debugger opens the file. If the source file is open, but not visible, RealView Debugger switches the code view to the source.

Break Sets a default breakpoint on the function using the `BREAKINSTRUCTION` command (see *Setting a default breakpoint on a function* on page 8-8).

If you want to set a non-default breakpoint on a function, that is you want to set a breakpoint that has qualifiers or actions, see *Setting a non-default breakpoint on a function* on page 8-9.

Go until Sets a temporary breakpoint at the specified function. The program then executes from the current position of the PC. When execution reaches the breakpoint it stops.

The temporary breakpoint exists only for the duration of this run and so is not shown in the Break/Tracepoints pane. Similarly, there is no red breakpoint marker shown in the source file. If the program stops before it reaches the temporary breakpoint, you must reinstate it before restarting the run.

This is equivalent to the CLI command:

`go` *`function`*

For example, `go @dhrystone\\DHRY_1\Proc_2`.

Show Type Info

Prints the type information for the selected function in the **Cmd** tab of the Output pane. This is equivalent to the CLI command:

`printtype` *`function`*

For example, `printtype @dhrystone\\DHRY_1\Proc_2`.

Show Full Info

Submits a `CEXPRESSION` command to calculate the value of a given expression by calling the specified target function.

The function is converted into a debugger call macro, and strings and arrays passed to the target function are copied onto a stack maintained by RealView Debugger. A function called in this way behaves as though it had been called from your program.

Note

Target calls are not supported by all debug environments.

Results are displayed in either floating-point format, address format, or in decimal, hexadecimal, or ASCII format depending on the type of variables used in the expression.

Set PC to Function

Submits a `SETREG` command to set the PC register to the start address of the selected function. For example:

```
setr @PC=@dhrystone\\DHRY_1\Proc_1
```

The Code window scrolls to the specified function and the red box shows the location of the PC.

Scope To Locates the start of the function in the current view of the File Editor pane:

- If the **Dsm** tab is selected, the function is displayed in the disassembly view.
- If a source file tab is selected, for example `dhry_1.c`, and the chosen function is in that source file, that function is displayed.

  If the chosen function matches a source file that is not selected, RealView Debugger selects the source file. If the source file is not already open, then RealView Debugger opens it.

If the selected function is in an ARM library, then RealView Debugger is unable to display the source, and displays the following message in the **Src** tab:

`No source for context: module\function`

module`\`*function* is the function reference, for example, `STDIO\fgetc`.

In all cases, RealView Debugger also locates the function in the **Dsm** tab. Select the **Dsm** tab to view the start address for the function.

8.2.11 Working with the Variables tab

The **Variables** tab in the Data Navigator pane lists the variables in all functions by default, or in the module you selected from the **Modules** tab (see *Working with the Modules tab* on page 8-10). You can also limit the list of variables displayed by specifying a filter (see *Applying a filter* on page 8-6).

Variables tab operations

You can perform the following operations with the **Variables** tab:

- Sort the variables in ascending or descending order of Variable Name, Address, Scope, Module, or Image. To do this, click on the appropriate column header.
- Print the hexadecimal value of the variable to the **Cmd** tab of the Output pane. To do this, double-click on the variable entry.
- Various operations available from the **Variables** context menu. To display this context menu, right-click on a variable entry. See *Variables context menu* on page 8-15 for a description of these operations.

Variables context menu

The **Variables** context menu contains the following options:

Show Publics

When enabled (checked), lists all global or public variables with scope over all parts of the image, or selected module.

Show Statics When enabled (checked), lists all static variables in the image, or selected module.

Show Labels When enabled (checked), lists all code labels with scope over the entire image, or selected module.

Show Locals When enabled (checked), lists all local variables in the image, or selected module.

Show Library Symbols

When enabled (checked), lists the symbols in the libraries used by the image. The library symbols listed depends on the state of the **Show Publics**, **Show Statics**, **Show Labels**, and **Show Locals** options.

When disabled (unchecked), most library symbols are filtered out. This is the default.

Print Hex Prints the hexadecimal value of the selected variable in the **Cmd** tab of the Output pane. This is equivalent to the CLI command:

`print /h variable`

For example, `print /h @dhrystone\\Ch_1_Glob`.

You can also display the hexadecimal value by double-clicking on the variable entry.

Print Decimal

Prints the decimal value of the selected variable in the **Cmd** tab of the Output pane. This is equivalent to the CLI command:

`print variable`

For example, `print @dhrystone\\Ch_1_Glob`.

Watch Adds a watch for the selected variable.

Show Type Info

Prints the type information for the selected variable in the **Cmd** tab of the Output pane. This is equivalent to the CLI command:

`printtype variable`

For example, `printtype @dhrystone\\Ch_1_Glob`.

Show Full Info

Prints the full information for the selected variable in the **Cmd** tab of the Output pane. This is equivalent to the CLI command:

`printsym` *`variable`*

For example, `printsym @dhrystone\\Ch_1_Glob`.

Break... Displays the Set Address/Data Breakpoint dialog box, where you can set a hardware breakpoint (see *Setting a hardware breakpoint on a variable* on page 8-9).

8.3 Using the Symbol Browser pane

Use the Symbol Browser pane to examine C++ classes in your application program.

To examine C++ classes either:

- Select the menu option **View → Symbol Browser** from the Code window main menu.
- Move the focus to the chosen pane, for example the Watch pane, click the **Pane Control** menu, and select **New Pane → Symbol Browser** from the available options.

An example display is shown in Figure 8-2.

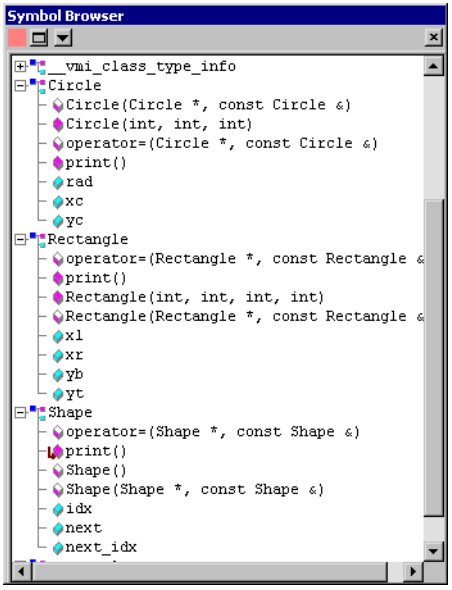

Figure 8-2 C++ symbols in the Symbol Browser pane

This section describes:

- *Viewing details of a class* on page 8-18
- *Viewing details of a function* on page 8-18.

8.3.1 Viewing details of a class

Colored icons are used to identify different components within a class:

Filled stack + arrow

This magenta icon indicates a function which is a declared member of the parent class.

Filled stack The magenta filled stack indicates that a member function of the parent class is both declared and defined. These members are real in that they are called during execution.

Hollow stack

Where magenta stacks are hollow, they indicate that a member function of the parent class is declared but not defined. These members are virtual in that they are not called during execution.

Filled block The filled blue block indicates a data object that is manipulated by a class function using operators, or methods, defined in the class.

Right-click on a chosen class to display the **Class** menu. This contains:

Find Class Definition...

Displays the Find in Files dialog box where you can specify the class details and so locate the definition in your source code.

Use this dialog box to locate class definitions when these are not provided by the compiler.

See the chapter that describes searching in *RealView Debugger v1.8 Project Management User Guide* for details on using the Find in Files dialog box.

Properties... Displays a text description of the item under the cursor.

8.3.2 Viewing details of a function

Right-click on a function to display the **Function** menu. This contains:

Show Function Definition

Scopes to the selected function. This option is enabled for defined functions identified by a magenta filled stack icon.

Set Break Sets a breakpoint at the selected function. This option is enabled for defined functions identified by a magenta filled stack icon.

Properties... Displays a text description of the item under the cursor.

8.4 Using the Function List dialog box

Use the Function browser to examine the different functions that make up your program.

This section describes:

- *Displaying the Function List dialog box*
- *Specifying the list*
- *Refining the list* on page 8-20
- *Selecting a function* on page 8-20
- *Closing the browser* on page 8-20.

8.4.1 Displaying the Function List dialog box

Display the Function List dialog box, shown in Figure 8-3, as described in *List browser dialog boxes* on page 8-2.

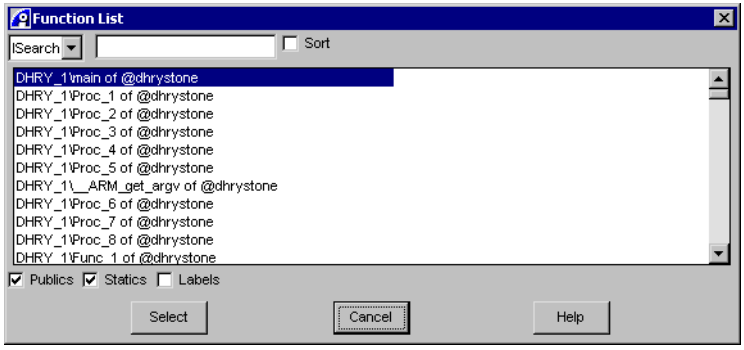

Figure 8-3 Function List dialog box

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
```

8.4.2 Specifying the list

When you first open the Function List dialog box, the list entries are determined by the default search entry **ISearch** but you can decide which functions are displayed by applying a search filter.

See *Specifying browser lists* on page 8-27 for details of how to specify the list for the chosen browser.

8.4.3 Refining the list

The Function List dialog box contains check boxes that enable you to refine what is displayed in the list box:

Publics Displays global or public functions with scope over all parts of the program.

Statics Displays static functions.

Labels Displays code labels with scope over the entire function.

8.4.4 Selecting a function

With a list of functions displayed in the Function List, select a function entry and click the **Select** button to select that function.

8.4.5 Closing the browser

When you click **Select**, the Function List dialog box closes automatically. Otherwise, click **Cancel** to exit the browser.

8.5 Using the Variable List dialog box

Use the Variables browser to examine the different variables used in your program.

This section describes:

- *Displaying the Variable List dialog box*
- *Specifying the list*
- *Refining the list* on page 8-22
- *Closing the browser* on page 8-22.

8.5.1 Displaying the Variable List dialog box

Display the Variable List dialog box, shown in Figure 8-4, as described in *List browser dialog boxes* on page 8-2.

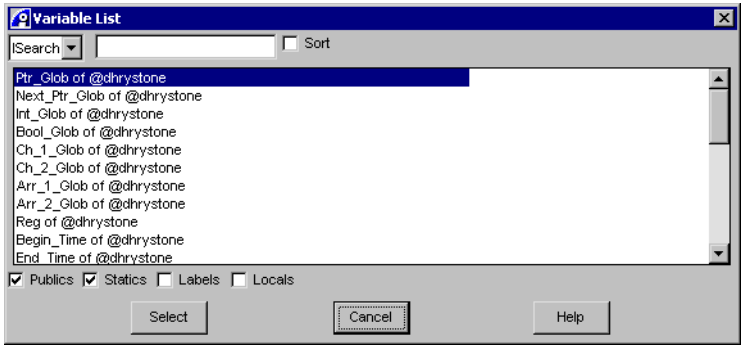

Figure 8-4 The Variable List

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, for example:

```
DHRY_2\Proc_8\Int_Index local of @dhrystone
```

8.5.2 Specifying the list

When you first open the Variable List dialog box, the list entries are determined by the default search entry **ISearch** but you can decide which variables are displayed by applying a search filter.

See *Specifying browser lists* on page 8-27 for details of how to specify the list for the chosen browser.

8.5.3 Refining the list

The Variable List dialog box contains check boxes that enable you to refine what is displayed in the list box:

Publics Displays global or public variables with scope over all parts of the program.

Statics Displays static variables.

Labels Displays code labels with scope over the entire function.

Locals Displays local variables with scope within the current function.

8.5.4 Closing the browser

When you click **Select**, the Variables List dialog box closes automatically. Otherwise, click **Cancel** to exit the browser.

8.6 Using the Module/File List dialog box

Using the Module/File browser enables you to examine the different files and modules that make up your program and how these components are accessed during program execution. In this way you can locate errors during your debugging session.

This section describes:

- *Displaying the Module/File List dialog box*
- *Specifying the list* on page 8-24
- *Selecting a module* on page 8-24
- *Closing the browser* on page 8-24.

8.6.1 Displaying the Module/File List dialog box

Display the Module/File List dialog box, shown in Figure 8-5, as described in *List browser dialog boxes* on page 8-2.

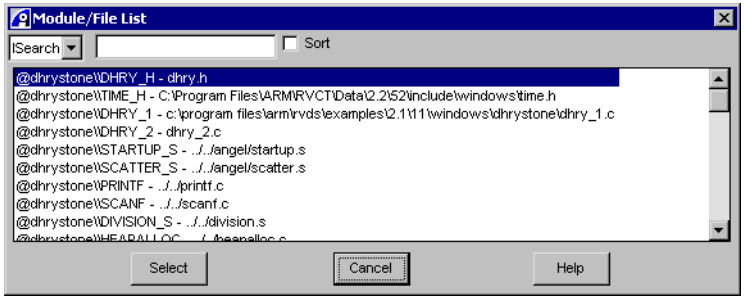

Figure 8-5 Module/File List dialog box

The Module/File List dialog box displays, in order of appearance, all the modules and files in the current program. Each entry in the list shows the module name and then the filename, if known, for example:

```
@dhrystone\\DHRY_2 - dhry_2.c
```

The program name is attached at the start using `@`, for example `@dhrystone\\`.

Module names qualify symbolic references. The module name is usually the filename without the extension. All module names are converted to uppercase by RealView Debugger. If the extension is not standard, the extension is preserved, and the dot is replaced with an underscore, for example `sample_arm.c` is converted to `SAMPLE_ARM`, and `sample_arm.h` is converted to `SAMPLE_ARM_H`.

If two modules have the same name then RealView Debugger appends an underscore followed by a number to the second module, for example `SAMPLE_1`. Additional modules are numbered sequentially, for example, if there is a third module this becomes `SAMPLE_2`.

Following this convention avoids any confusion with the C dot operator indicating a structure reference.

8.6.2 Specifying the list

When you first open the Module/File List dialog box, the list entries are determined by the default search entry **ISearch** but you can decide which modules and files are displayed by applying a search filter.

See *Specifying browser lists* on page 8-27 for details of how to specify the list for the chosen browser.

8.6.3 Selecting a module

To select a module:

1. Display the Module/File List dialog box as described in *List browser dialog boxes* on page 8-2.
2. Select a module from the list.
3. Click **Select** to select that module.

8.6.4 Closing the browser

When you click **Select** the Module/File List dialog box closes automatically. Otherwise, click **Cancel** to exit the browser without adjusting the scope.

8.7 Using the Register List Selection dialog box

The Register List Selection dialog box enables you to select a register from a list of memory mapped registers for the current target connection. The list of registers is generated from those defined in the `Peripherals` group of the Board/Chip Definition file assigned to the connection. For more details, see the *RealView Debugger v1.8 Target Configuration Guide.*

Note

You must assign a Board/Chip Definition file (for example, `AP.bcd`) to the connection to use this dialog box.

This section describes:

- *Displaying the Register List Selection dialog box*
- *Selecting a register* on page 8-26
- *Closing the browser* on page 8-26.

8.7.1 Displaying the Register List Selection dialog box

Display the Register List Selection dialog box, shown in Figure 8-6, as described in *List browser dialog boxes* on page 8-2.

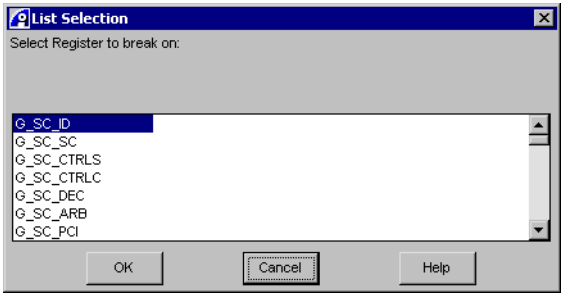

Figure 8-6 Register List Selection dialog box

The Register List Selection dialog box displays, in alphabetical order, the memory mapped registers. These registers are defined any Board/Chip Definitions you have assigned to the current target connection. The registers shown in Figure 8-6 are those for the Integrator/AP board, which are defined in the `AP.bcd`. For more details, see the *RealView Debugger v1.8 Target Configuration Guide.*

8.7.2 Selecting a register

To select a register:

1. Display the Register List Selection dialog box as described in *List browser dialog boxes* on page 8-2.
2. Select a register from the list.
3. Click **OK** to select that register.

8.7.3 Closing the browser

When you click **OK** the Register List Selection dialog box closes automatically. Otherwise, click **Cancel** to exit the browser without adjusting the scope.

8.8 Specifying browser lists

When you first open a browser, the list entries shown in the dialog box are determined by the default search entry **ISearch** but you can decide what contents are displayed by applying a search filter. To change the search mechanism either:

- click on the drop-down arrow to display the search list and select a search
- highlight the contents, for example **ISearch**, and press F or I to toggle the contents.

This section describes:

- *Specifying a list*
- *Applying a filter* on page 8-28.

8.8.1 Specifying a list

When you change the browser search mechanism to specify a new list, RealView Debugger performs the match against list entry *segments* and not against the text string as shown in the display list. This applies if you choose to sort the display list (see the description of **Sort** in this section) or to apply a filter to limit the search (see *Applying a filter* on page 8-28 for details).

To specify the search:

List of entries displayed

The list of entries displayed can be specified using one of the following:

ISearch With this selected, the list of entries is created using the default search mechanism based on the names of the modules, files, functions, or variables found in your program. Enter a partial name in the Text entry field to move the highlight to the first matching occurrence. The search is case insensitive.

Filter Use a filter to limit the search to list only those symbols that match certain criteria, see *Applying a filter* on page 8-28 for details.

Text entry field

Enter here the filter to be used in the search and then press Enter. The filter enables you to search using the metacharacters listed in Table 8-1 on page 8-7. You can enter a list of search rules by combining different operators, see *Applying a filter* on page 8-28 for details of how to apply a search filter.

Sort Select this check box to order the display list in alphabetical order based on the names of the modules, files, functions, or variables found in your program. The sort is case insensitive, that is uppercase and lowercase are treated as identical. `Sort` is unchecked by default.

Note

If you change the default search entry to **Filter** and then enter a filter to set the display criteria, these settings are maintained when you close, or cancel, the browser.

8.8.2 Applying a filter

Using a filter enables you to narrow down the search when displaying the list of modules, files, functions, or variables. Table 8-1 on page 8-7 shows the metacharacters you can use to specify the filter rule or rules. When entering a filter, characters are case sensitive, for example the filter `*DHRY*` returns a list of three modules but `*dhry*` returns an empty list.

When you have completed the filter, press Enter and the list is refreshed. By repeatedly entering filters, and pressing Enter, you can refine the search to focus on selected modules, files, functions or variables.

8.9 Using the Favorites Chooser/Editor dialog box

This section describes how to use the Favorites Chooser/Editor dialog box. It includes:

- *Overview of the Favorites Chooser/Editor dialog box*
- *Displaying the Favorites Chooser/Editor dialog box* on page 8-30
- *Favorites Chooser/Editor dialog box interface components* on page 8-30
- *Favorites categories used by RealView Debugger features* on page 8-32.

8.9.1 Overview of the Favorites Chooser/Editor dialog box

The Favorites Chooser/Editor dialog box enables you to save expressions that you use on a regular basis to a personal Favorites List. The following types of expression can be set as favorites:

- addresses and address ranges
- data values
- breakpoints and tracepoints
- qualifiers and actions for breakpoints
- watch expressions
- directories
- filenames.

When you first start to use RealView Debugger on Windows, your personal Favorites List is empty. You can create expressions and add them directly to this list or you can add expressions that you have been using in the current debugging session.

If you create an expression manually, you must make sure that the expression matches the Favorites Category (see *Favorites Chooser/Editor dialog box interface components* on page 8-30). For example, you must add an address range to the Addresses category, but not to the Break/Tracepoints category. If you do add a favorite to the wrong category, then RealView Debugger might not behave as expected, or an error message might be displayed.

RealView Debugger keeps a record of all favorite expressions that you set during your debugging session as part of your history file `exphist.sav`, which is located in your RealView Debugger home directory. This file also contains a history of other entities, for example files and directories you have accessed during the debugging session. By default, at the end of your debugging session, these processor-specific lists are saved in the history file.

Note

You must connect to a target to enable RealView Debugger favorites.

8.9.2 Displaying the Favorites Chooser/Editor dialog box

There are many ways you can display the Favorites Chooser/Editor dialog box in RealView Debugger:

- From any dialog boxes that have a drop-down arrow to the right of the various fields, for example, the various breakpoint and tracepoint dialog boxes.
- By selecting the appropriate favorites option from a menu, for example, the **Set from Favorites** option on the context menu of the Register pane.

The Favorites Chooser/Editor dialog box is shown in Figure 8-7.

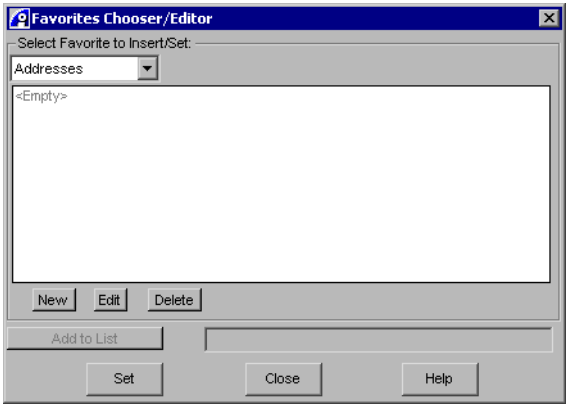

Figure 8-7 Favorites Chooser/Editor dialog box

8.9.3 Favorites Chooser/Editor dialog box interface components

The main interface components of the Favorites Chooser/Editor dialog box (see Figure 8-7) are:

Favorites Category

This is the category that is to contain the favorite expression. *Favorites categories used by RealView Debugger features* on page 8-32 identifies the dialog boxes where each categories are used.

RealView Debugger automatically sets this to the context from which you displayed the dialog box. For example, if you want to set a breakpoint from the Break/Tracepoint favorites, RealView Debugger sets this to Break/Tracepoints.

Alternatively, you can select another category if you want to add an expression manually. Use the **New** button to create a new favorite yourself.

Favorites List

Lists any favorites that are currently defined for the selected Favorites Category.

New Click this button the manually create a new favorite. When you click this button a New/Edit Favorite dialog box is displayed. Enter the required expression and a meaningful description. The expression must match the Favorites Category, for example, a breakpoint if the Break/Tracepoint category is selected. When you have finished, click **OK** to add the expression to the Favorites List.

Edit Click this button to edit the selected favorite.

Delete Deletes the selected favorite from the Favorites List.

Add to List This button is enabled if an expression is shown in the text box to the right of the button. Otherwise it is disabled.

When you click this button a New/Edit Favorite dialog box is displayed showing the expression from the text box. You can also enter a meaningful description for the expression. When you have finished, click **OK** to add the expression to the Favorites List.

Set Closes the dialog, and sets the selected favorite for the current context.

Close Closes the dialog box without using a favorite.

Help Displays online help for this dialog box.

8.9.4 Favorites categories used by RealView Debugger features

Table 8-2 describes where each Favorites Category is used in RealView Debugger.

Table 8-2 Favorites Categories used by RealView Debugger features

Favorites Category	Where it is used in RealView Debugger
Addresses	This is used where you can specify an address or an address range: • In the dialog boxes for setting breakpoints, where you want to specify an address. See Chapter 4 *Working with Breakpoints* for details on using the breakpoint dialog boxes. • In the dialog boxes for setting tracepoints, where you want to specify an address or an address range. See the chapter the describes tracing in RealView Debugger in the *RealView Debugger v1.8 Extensions User Guide* for details on using the tracepoint dialog boxes. • In the dialog boxes used to locate a specific address, such as in the: — Stack pane (see *Using the Stack pane* on page 6-31) — Memory pane (see *Displaying memory contents* on page 6-19) — Dsm tab.
Data Values	Used to specify data values in the dialog boxes when setting: • Breakpoints See Chapter 4 *Working with Breakpoints* for details on using the breakpoint dialog boxes. • Tracepoints See the chapter the describes tracing in RealView Debugger in the *RealView Debugger v1.8 Extensions User Guide* for details on using the tracepoint dialog boxes. • The value of a memory location in the Memory pane. See *Displaying memory contents* on page 6-19. • The value of a register in the Register pane. See *Changing register contents* on page 6-6.
Data Ends	Used by the HW Break on Data Value match dialog box, to specify a data value modifier. See *Using the HW Break on Data Value match dialog box* on page 4-29 for details.

Table 8-2 Favorites Categories used by RealView Debugger features (continued)

Favorites Category	Where it is used in RealView Debugger
Watch Expressions	Used for specifying watchpoints in the Watch pane. See *Working with watches* on page 6-40 for details on setting watchpoints.
Break/Tracepoints	This is used to specify CLI commands for setting breakpoints and tracepoints: • For details on setting breakpoints, see Chapter 4 *Working with Breakpoints*. • For details on setting tracepoints, see the chapter the describes tracing in RealView Debugger in the *RealView Debugger v1.8 Extensions User Guide*.
Break Qualifiers	This is used for breakpoint qualifiers on the Set Address/Data Breakpoint dialog box. See *Specifying Qualifiers* on page 4-52 for details.
Break Actions	This is used for breakpoint actions on the Set Address/Data Breakpoint dialog box. See *Specifying Actions* on page 4-53 for details.
Filenames	This is used for the various open and save file dialog boxes to specify files, such as opening source files, include and workspace files, and selecting log files.
Directories	This is used for the various open and save file dialog boxes to specify directories, such as opening source files, include and workspace files, and selecting log files.

Chapter 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, and how to save and edit your macros. It contains the following sections:

9.1 About macros

Macros are interpreted C code running on the host with access to target memory and symbols, user-defined debugger symbols (in host or target memory), and debugger functions. Macros can access debugger variables, external windows and programs, and can be attached to breakpoints, aliases, and windows.

A macro can contain:

- a sequence of expressions
- string formatting controls
- statements
- calls to other macros
- predefined macros
- target functions
- debugger commands.

You can define and use macros at any time during a debugging session to use the commands or statements contained in the macro. You call the macro with a single command using the name. The macro definition might contain parameters that you change each time the macro is called.

When a macro is defined, you can use it as:

- a complex command or in an expression
- an attachment to a breakpoint to create breakpoint condition testing
- an attachment to a window or file where the macro can send information.

Note

After a macro has been loaded into RealView Debugger, the definition is stored in the symbol table. If the symbol table is recreated, for example when an image is loaded with symbols, any macros are automatically deleted. Disconnecting also clears any macros.

This section gives an overview of macros in RealView Debugger. It includes the following sections:

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

RealView Debugger enables you to enter debugger commands on the command line, or when using the headless debugger, for example `PRINTF` or `SETMEM`.

You can define a macro that contains a sequence of debugger commands. When used in this way, the command must be enclosed by dollar signs (`$`), shown in Example 9-1.

Example 9-1 Using debugger commands in macros

```
some_commands
define int registers()
{
some_commands
for(macro_index = 0; macro_index < 6; macro_index++)
  {
    macro_bin_str[macro_index + 8] = macro_cpsr_mode[macro_index +
(6*macro_cpsr_key)];
  }
```

```
  macro_bin_str[14] = 0x00;

  $printf "  r0 = 0x%x" ,@r0$;
  $printf "  r1 = 0x%x" ,@r1$;
  $printf "  r2 = 0x%x" ,@r2$;
some_commands
}
.
```

Macros containing commands are similar to command files and can be used for setting up complex initialization conditions. These macros are executed by entering the macro name and any parameters on the RealView Debugger command line.

The following commands cannot be used inside a macro:

- ADD
- DEFINE (unless it is the macro definition itself)
- DELETE
- HELP
- HOST
- INCLUDE
- QUIT.

Because macros can return a value, they can also be used in expressions. When the macro executes, the return value is used according to the return type.

For full details on all CLI commands, see *RealView Debugger v1.8 Command Line Reference Guide*.

Attaching macros

Macros can also be invoked as actions associated with:

- a window
- a breakpoint
- deferred commands, for example BGLOBAL.

In this case, execution-type commands cannot be used inside a macro:

- GO
- GOSTEP
- STEPLINE, STEPO
- STEPINSTR, 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 a text editor and load the macro definition file in with the `INCLUDE` command. The number of macros that can be defined is limited only by the available memory on your workstation.

You can also create, edit, save, and delete macros from the Code window using the **Debug** menu, as demonstrated in *Using macros* on page 9-8.

Note

After a macro has been loaded into RealView Debugger, the definition is stored in the symbol table. If the symbol table is recreated, for example when an image is loaded with symbols, any macros are automatically deleted. Disconnecting also clears any macros.

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

The type of the macro return value is specified by `return_type` when you define the macro. If `return_type` is not specified, then type `int` is assumed.

When attached to a breakpoint, the macro return value controls the behavior of the breakpoint, see *Using macros with breakpoints*.

9.1.6 Using macros with breakpoints

When you set a breakpoint, you can also associate a macro with that breakpoint to set up complex break conditions. When you attach a macro to a breakpoint, it acts as a condition qualifier (see *Setting conditional breakpoints* on page 4-34). Therefore, you can test your program variables and decide how the breakpoint behaves when it is hit (see *Controlling breakpoint behavior with macro return values* on page 9-7).

The frequency that a macro runs when the breakpoint is hit depends on the order it appears with other condition qualifiers. See *Controlling the behavior of breakpoints* on page 4-59 for more details.

You can use conditional statements in your macro to change the execution path when the breakpoint is triggered depending on variables on the debug target system or on the host. This enables you to control program execution during your debugging session or when there is no user intervention.

You can also use high-level expressions in macros. Combining these conditional statements and expressions enables you to patch your source program.

Breakpoint macros can be used to fill out stubs, such as I/O handling, and also to simulate complex hardware.

For an example of attaching a macro to a breakpoint and using it to control program execution see *Attaching macros to breakpoints* on page 4-56.

Note

You cannot use the `GO`, `GOSTEP`, `STEPINSTR`, `STEPLINE`, `STEPO`, or `STEPOINSTR` commands in a macro that is attached to a breakpoint. If any of these commands are present in a such a macro, the following messages are displayed:


```
Error: Cannot perform operation - thread is running.
Error: E0081: Runtime error in macro.
```

Controlling breakpoint behavior with macro return values

You can use macro return values to control what action RealView Debugger takes when a breakpoint is triggered:

- If the macro returns a nonzero value, RealView Debugger continues program execution. Any qualifiers and actions that appear after the macro are not processed, and the triggering of the breakpoint is not recorded.
- If a macro returns a value of zero, RealView Debugger stops program execution. Any actions that are assigned to the macro are performed.

Note

This behavior might be different if you assign additional condition qualifiers to the breakpoint. See *Setting conditional breakpoints* on page 4-34 for more details.

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 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 9-10
- *Testing a macro* on page 9-10
- *Editing a macro* on page 9-11
- *Copying a macro* on page 9-13
- *Deleting a macro* on page 9-13
- *Calling a macro* on page 9-14.

9.2.1 Creating a macro

To create a macro from within RealView Debugger, you can use the Add/Edit Macros dialog box. It is not possible to create macros using CLI commands on the command line.

Complete the following steps to create a macro for use with a debug target:

1. Connect to your target and load an image, for example `dhrystone.axf`.
2. Select **Edit → Advanced → Show Line Numbers** to display line numbers.

   This is not necessary but might help you to follow the examples.
3. Select **Tools → Add/Edit Debugger Macros...** to display the Add/Edit Macros dialog box.

   When you open this dialog box, the Existing Macros display list is empty, unless you have previously loaded macros into RealView Debugger.

   The Macro Entry Area gives advice on how to use the buttons, **New**, **Show**, and **Copy**.

   This area shows the definition of the macro when it has been created.
4. Click **New** to create the macro. This inserts the default name `int Macro()` in the Name data entry field and inserts `{}` in the Macro Entry Area ready for editing.
5. Edit the default macro name so that it shows `int tutorial(var1)`.
6. Enter the macro contents to show:

```
int var1;

{
$printf “value=%d\n”,var1$;
}
```


When creating a macro, variables must be declared at the start of the macro definition. This also applies to macros that you create using a text editor.

The Add/Edit Macros dialog looks like the example shown in Figure 9-1.

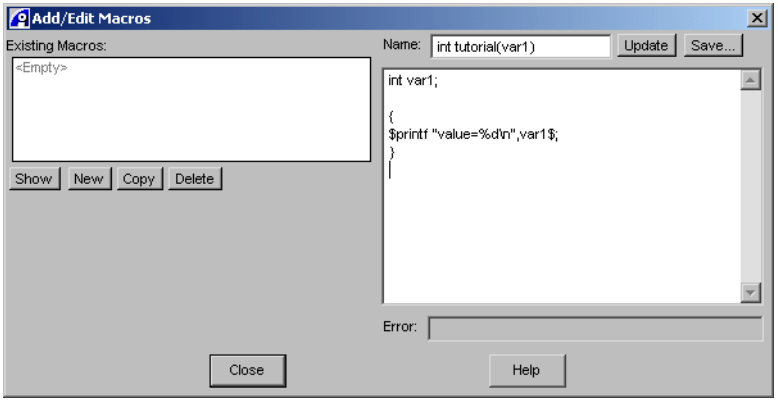

Figure 9-1 Creating a macro

The macro uses the `PRINTF` command and so the command must be enclosed by dollar signs (`$`), shown in the Macro Entry Area.

7. Click **Update** to pass the macro definition to RealView Debugger, where it is stored in the symbol table. This adds the new macro to the Existing Macros display list.

   If there are any errors in the macro text, you are notified when you try to pass the macro to RealView Debugger.

   Note

   If the symbol table is recreated, for example when an image is loaded with symbols or if you disconnect, any macros are automatically deleted.

8. View the Output pane message:

   ```
   def int tutorial(var1)
   ```

9. Click **Save...** to display the Select file to save/append into dialog box where you can choose where to save the new macro definition file for later re-use.

   The macro name has been entered automatically with the extension `.inc`.

10. Save the new macro, with the default name, in your `\home` directory.

    If the chosen file already exists, RealView Debugger displays a warning message and gives you the option to append the new file or to overwrite the existing contents.

11. Click **Close** to close the Add/Edit Macros dialog box and return to the Code window.

9.2.2 Viewing a macro

View the contents of a macro using:

- the File Editor pane
- an external text editor
- the `SHOW` command.

Viewing a macro using any of these methods differs from viewing the macro contents in the Add/Edit Macros dialog box:

- The Macro Entry Area, in the dialog box, does not show the macro `DEFINE` command or the terminator (a period used as the first and only character on the last line).
- The `SHOW` command does not display the terminator.

Note

If you are using a predefined macro, you cannot use the `SHOW` command to view the definition.

9.2.3 Testing a macro

To test the macro you have created, you execute the loaded image and then enter the macro on the RealView Debugger command line:

1. Click on the **Src** tab to view the source file `dhry_1.c`.
2. Set a default breakpoint by double-clicking on line 187.
3. Click **Go** to start execution. When asked for the number of runs, use a small number, for example `5000`.
4. When the program reaches the breakpoint, enter the following command and press Enter:

   ```
   tutorial(Int_2_Loc)
   ```

   This displays the current value of the variable `Int_2_Loc` in the Output pane.
5. Step through the program a few more times using the macro to monitor the variable. You can use the up arrow to step back through the commands already submitted on the command line.

9.2.4 Editing a macro

To edit the macro that you have created and retest it to verify the changes:

1. Select **Tools → Add/Edit Debugger Macros...** to display the Add/Edit Macros dialog. The Existing Macros display list shows the `tutorial` macro you created and it is highlighted.

2. Click **Show** to see the contents of the macro.

3. Change the macro name to read:

   ```
   int tutorial(var1,var2,var3)
   ```

4. In the Macro Entry Area change the variable definition to read:

   ```
   int var1;
   int var2;
   int var3;
   ```

5. In the Macro Entry Area change the body of the macro to read:

   ```
   var1==var1++;
   $fprintf 250,  "value=%d\n",var1$;
   $fprintf 250,  "value=%d\n",var2$;
   $fprintf 250,  "value=%d\n",var3$;
   ```

   This change increments `var1` and displays the output of the macro in a window instead of the Output pane.

6. In the Macro Entry Area add this line at the end of the macro:

   ```
   return(var1);
   ```

   This change causes the macro to return the value of the variable. If the value is nonzero, then RealView Debugger continues program execution after reporting the result. If the value returned is zero, then execution stops.

   The macro now looks like the one shown in Figure 9-2 on page 9-12.

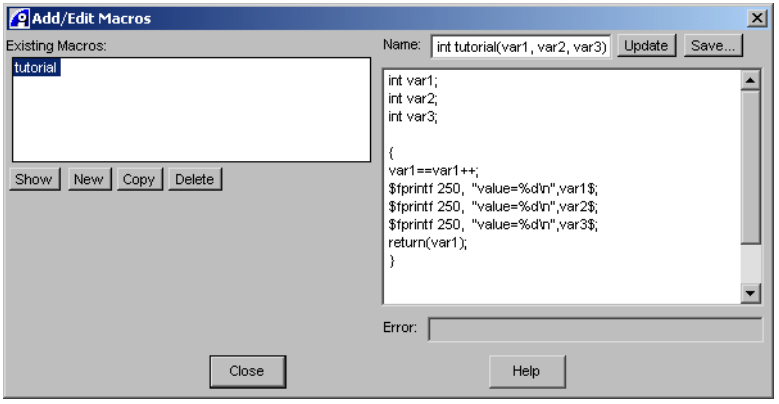

Figure 9-2 Editing a macro body

7. Click **Update** to pass the macro definition to RealView Debugger.
8. View the Output pane message:

   ```
   def int tutorial(var1, var2, var3)
   ```

9. Click **Save...** to save the updated macro in the same location. This generates a prompt to enable you to Append or Replace the existing file. Click **No** to replace the existing `tutorial.inc`.
10. Click **Close** to close the Add/Edit Macros dialog.

To test the macro you have created, you execute the image and then enter the macro on the RealView Debugger command line:

1. Select **Target → Reload Image to Target** to reload the image to your debug target.
2. Enter this command:

   ```
   VOPEN 250
   ```

   This opens a window ready to display the results returned from the macro.
3. If you no breakpoint is set on line 187, double-click on line 187 to set a default breakpoint.
4. Click **Run** to start execution. When asked for the number of runs, use a small number, for example `5000`.
5. When the program reaches the breakpoint, enter the following command on the command line of the Code window and press the Enter key:

   ```
   tutorial(Int_1_Loc, Int_2_Loc, Int_3_Loc)
   ```


This displays the current value of the variables in the window.

6. Step through the program a few more times using the macro to monitor the variables. You can use the up arrow to step back through the commands already submitted on the command line.

Remember to save your macros before you exit RealView Debugger. There is no warning if any macro definitions are unsaved.

9.2.5 Copying a macro

You can use an existing macro to form the basis of a new macro:

1. Select **Tools → Add/Edit Debugger Macros...** to display the Add/Edit Macros dialog and so edit the macro. The Existing Macros display list shows the `tutorial` macro you created and it is highlighted.
2. Click **Show** to see the contents of the macro.
3. Click **Copy**. This automatically adds an integer number to the end of the macro name in the Name field, starting at one. For example, `int tutorial1(var1,var2,var3)`. Subsequent copies are numbered sequentially, for example, `int tutorial2...` and `int tutorial3...`. However, you can change the this name to your own choice.
4. In the Macro Entry Area change the body of the macro as required.
5. Click **Update** to pass the macro definition to RealView Debugger.
6. View the Output pane message, assuming that you do not change the default name:

   ```
   def int tutorial1(var1,var2,var3)
   ```

7. Click **Save** to save the updated macro in the usual way.
8. Click **Close** to close the Add/Edit Macros dialog.

The number of macros that can be defined is limited only by the available memory on your workstation.

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 only delete one macro at a time in this way.

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.

After a macro has been loaded into RealView Debugger, the definition is stored in the symbol table. If the symbol table is recreated, for example when an image is loaded with symbols, any macros are automatically deleted.

Note

Reloading an image with all associated symbols also deletes any macros.

To load, or reload, macros into RealView Debugger:

1. Select **Tools** from the main menu to display the **Tools** menu.
2. Select **Include Commands from File...** to display the Select File to Include Commands from dialog box.
3. Highlight the required `.inc` file and then click **Open**. This loads the selected macro into RealView Debugger.

   If there is an error in the `.inc` file, an error message is generated in the Output pane and the macro is undefined.
4. Select **Tools** → **Add/Edit Debugger Macros...** to display the Add/Edit Macros dialog box where your macro is now shown in the Existing Macros display list.

You can load in several macros in this way ready for use in your debugging session. When a macro is displayed in the Add/Edit Macros dialog box, it can be changed as described previously and re-used, and resaved if required.

Loading macros on connection

You can load one or more macros automatically when you connect to a given target. Do this by specifying the macros to be loaded using the Connection Properties window. Enter the calling command using the `Commands` setting in the `Advanced_Information` block, shown in Figure 9-3 on page 9-15.

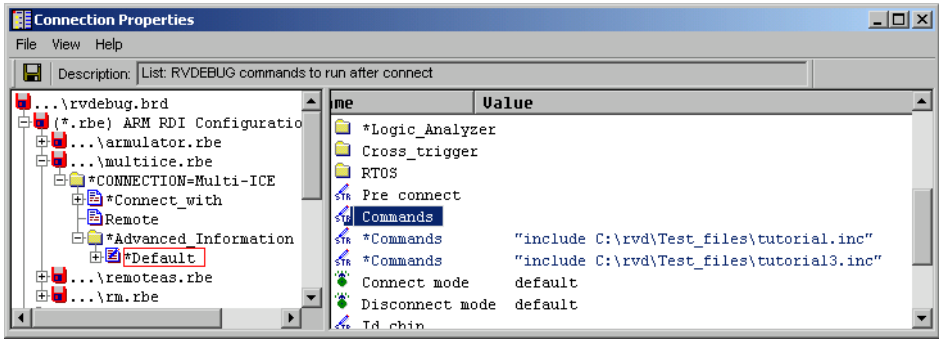

Figure 9-3 Loading macros on connection

If RealView Debugger cannot locate one of the specified files, it displays a message box and gives you the option to abort the connection.

For full details on how to specify these commands using the Connection Properties window, see the chapter that describes how to configure custom connections in *RealView Debugger v1.8 Target Configuration Guide*.

Loading macros from a project

You can load one or more macros automatically when you open a project that binds to a connection. Do this by specifying the macros to be loaded using the Project Properties window. Enter the calling command using the `Open_conn` setting in the `Command_Open_Close` group, shown in Figure 9-4.

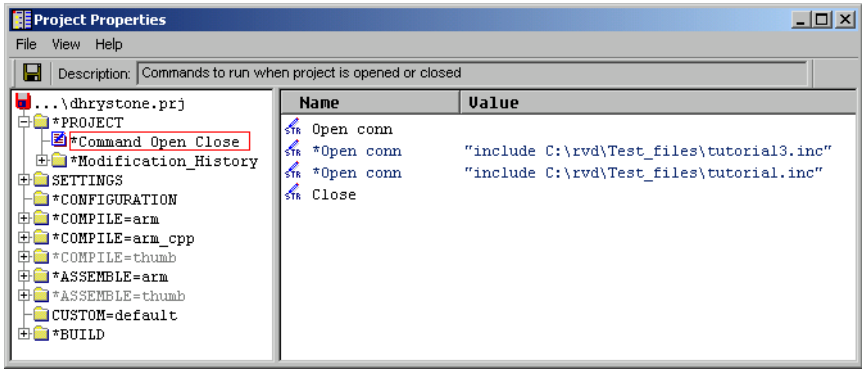

Figure 9-4 Loading macros from a project

If RealView Debugger cannot locate one of the specified files, it displays a message box. The project then opens and binds as normal.

This might be useful if combined with the `Open_load` setting in the `SETTINGS` group, for example to load the associated image and all symbols when the project binds.

For full details on how to specify these commands using the Project Properties window, see the chapter that describes how to customize your projects in *RealView Debugger v1.8 Project Management User Guide*.

9.3 Getting more information

See *RealView Debugger v1.8 Command Line Reference Guide* for more information on how to use macros:

- see the chapter that describes working with the CLI for details on writing your own scripts
- see the chapter that describes RealView Debugger commands for full details on using the `DEFINE` command
- see the chapter that describes predefined macros for full details on predefined and user interaction macros in RealView Debugger.

Chapter 10
Configuring Workspace Settings

This chapter explains how to use workspaces in RealView® Debugger, and describes how to configure your workspace settings. Read this chapter in conjunction with Appendix A *Workspace Settings Reference* that contains a detailed description of the workspace settings.

This chapter contains the following sections:

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
- *Opening workspaces* on page 10-5
- *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 v1.8 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="D:\ARM\RVD\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 10-5 for details.

If you are already using a workspace, select this option to close the current workspace before the new workspace opens, see *Closing workspaces* on page 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.

Save As Workspace...

Saves the current workspace to disk using a new name. This is useful if you have made changes since your debugging session began and want to save this new setup in a new workspace. This displays a dialog box where you can specify the new filename, for example `test_workspace.aws`.

The newly-specified workspace becomes the current workspace.

Workspace Options...

Displays the Workspace Options window where you can view the current workspace settings or make changes. See *Viewing workspace settings* on page 10-9 and *Configuring workspace settings* on page 10-15 for more details.

Save Settings on Exit

Selected by default, this option saves selected user-configured settings and the current workspace when you exit RealView Debugger. This enables you to start your next debugging session in the same state.

If selected, settings are saved in your start-up file, using the `.sav` extension by default, for use next time and the current workspace file is updated.

Same Workspace on Startup

Selected by default, this option saves the current workspace pathname in your `.sav` file so that the same workspace is used at the next start-up.

You can unselect this option so that the current workspace is not opened by default when you next start RealView Debugger. Unless you specify a workspace on the command line, RealView Debugger then runs without a workspace.

Close Workspace

Closes the current workspace. After the workspace closes, RealView Debugger displays a list box so you can close any open objects. See *Closing workspaces* on page 10-5 for more details.

Recent Workspaces

Displays the Recent Workspace menu that lists any previous workspaces that you have saved. Select a workspace from this menu to use those workspace setting.

10.1.3 Opening workspaces

If you open a new workspace, RealView Debugger adds the objects specified in the new workspace to all existing objects. This usually means that at least one more Code window opens on your desktop.

If you open a new workspace, you might see many new Code windows on your desktop. To avoid this, close open windows before opening the new workspace, see *Closing workspaces* for details.

When you open a new workspace, it might contain settings that override the current configuration. Where there is a conflict, a warning message is displayed and the new workspace settings are used.

10.1.4 Closing workspaces

You can explicitly close your current workspace, either:

- select **File → Workspace → Close Workspace**
- select **File → Workspace → Open Workspace...** to close the current workspace before the new one opens.

If you close your current workspace, the following applies:

- The contents of the default Code window do not change.
- If there are open objects, these do not change (see *Closing objects* for details).
- Any open objects are saved in the workspace before it closes so that they can be re-used when it next opens.
- If there are no open objects, the current workspace closes immediately.
- If the Workspace Options window is open, this closes automatically before the current workspace closes. If you have changed any workspace settings, these are saved.
- If the Options window is open, this is not affected when the workspace closes.

Closing objects

If you close a workspace, RealView Debugger enables you to close any open objects. 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 the open objects you want to close, shown in Figure 10-1.

Note

The Close Open Objects selection box is not displayed if there are no open objects.

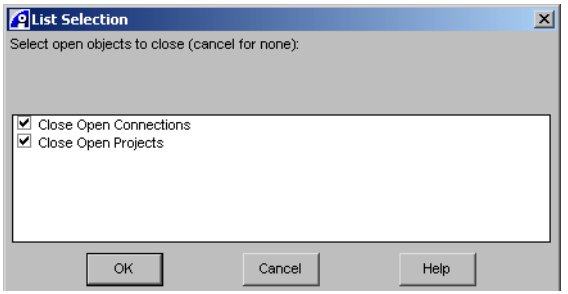

Figure 10-1 Close Open Objects selection box

The display list shows the open objects, that is:

- connections to debug targets
- any Code windows open in addition to the default Code window
- open projects including user-defined projects and auto-projects (see *Projects in workspaces* on page 10-6 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**.

10.1.5 Projects in workspaces

RealView Debugger saves a project load list when the current workspace closes. This is a list of open projects maintained when the debugger starts with this workspace or when you open this workspace in a session. This list includes user-defined projects and any auto-projects where you have saved the settings.

Where you are using a project load list, you must be aware of the following when the workspace opens:

- Where a project was bound to a connection when the workspace closed, binding details are saved and this is maintained when the connection is restored.
- Project binding details saved in the workspace take precedence. This applies even where the load list opens an *autobound* project that is unbound, that is where you have defined a `Specific_device` setting.
- Where there is no connection, the order in which projects open defines the active project.

If you are already running a debugging session and you open a new workspace, the project environment might change depending on the project load list (if any). RealView Debugger forces the project binding as defined in the workspace even if this means unbinding open projects. This is the case even if the open projects include an autobound project, that is a project where you have defined a `Specific_device` setting.

If the workspace opens with no saved binding details, the current project environment does not change. This is the case even if the project load list contains a project where you have defined a `Specific_device` setting.

Note

There is no warning when the project environment changes as a result of opening a workspace into a debugging session.

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 Code windows and their attachment.

See *RealView Debugger v1.8 Project Management User Guide* for full details on working with projects and project binding.

10.1.6 Creating an empty workspace

You can create a blank workspace settings file at any point during your debugging session. To do this, select **File → Workspace → New Workspace...** from the Code window main menu. This displays the New Workspace dialog box.

Use this to create an empty file, for example `New_workspace.aws`, in the new location. This becomes the current workspace. You must save settings to this new workspace settings file if you want it to be available at the next start-up.

If you are already working with a workspace this closes and then the new workspace opens ready for you to use. This does not override the current configuration.

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 10-10
- *Groups and settings* on page 10-13.

10.2.1 Using Settings windows

To access your current workspace settings or global configuration options:

File → Workspace → Workspace Options...

Displays the Workspace Options window where you can view the current workspace settings or make changes.

Tools → Options...

Displays the Options window where you can view the global configuration options, used by the current workspace, or make changes.

You have access to the global options when you are working without a workspace.

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.

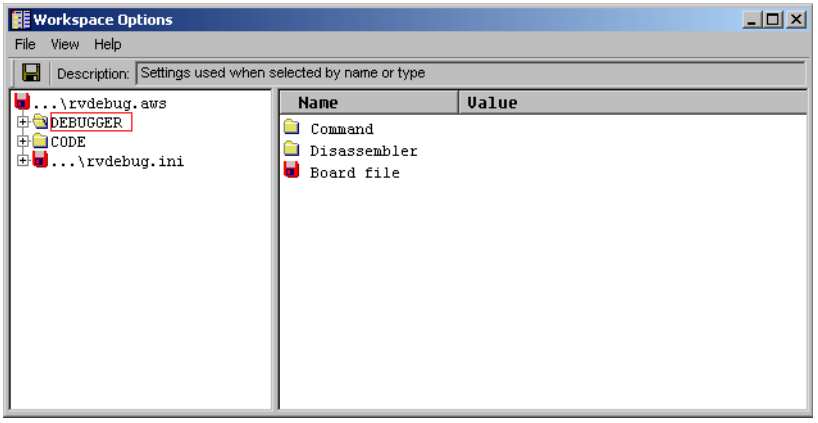

Figure 10-2 Workspace Options window

The main interface components of this window are:

Main menu This contains:

- **File** Displays the **File** menu where you can save the workspace file after you have made changes.
- **View** Displays the **View** menu to toggle the display to show all the settings or only those that have been edited.
- **Help** Displays the online **Help** menu.

Save icon Click this icon to save the workspace settings file to disk. The name of the current file is shown as the first entry in the left pane.

Description This field displays a one-line description about an entry selected in the List of Entries pane or Settings Values pane.

List of Entries pane

The left pane is the List of Entries pane, and shows configuration entries as a hierarchical tree with node controls (see *List of Entries pane* on page 10-11).

Settings Values pane

The right pane is the Settings Values pane, and shows the settings for the group selected in the List of Entries pane (see *Settings Values pane* on page 10-12).

When you close down RealView Debugger, your workspace settings file is updated with the current configuration, for example projects, connections, and open windows.

Note

See the RealView Debugger online help topic *Changing Settings* for details on all the entries in the Workspace Options window.

List of Entries pane

The left pane of the Workspace Options window, the List of Entries pane, shows workspace entries as a hierarchical tree with node controls (see Figure 10-3 on page 10-12). Groups of settings are associated with an icon to explain their function:

Red disk This is a container disk file.

RealView Debugger uses this, for example, to specify an include file.

Yellow folder

This is a parent group containing other groups (*rules pages*) and/or entries.

Rules page A rules page is a container for settings values that you can change in the right pane. When unselected, the pencil disappears (see Figure 10-3 on page 10-12).

This icon only appears in the left pane.

An asterisk (*) is placed at the front of an entry to show that it has changed from the default or was created by RealView Debugger. Figure 10-3 on page 10-12 shows an example settings file that specifies a multiprocessor debugging session.

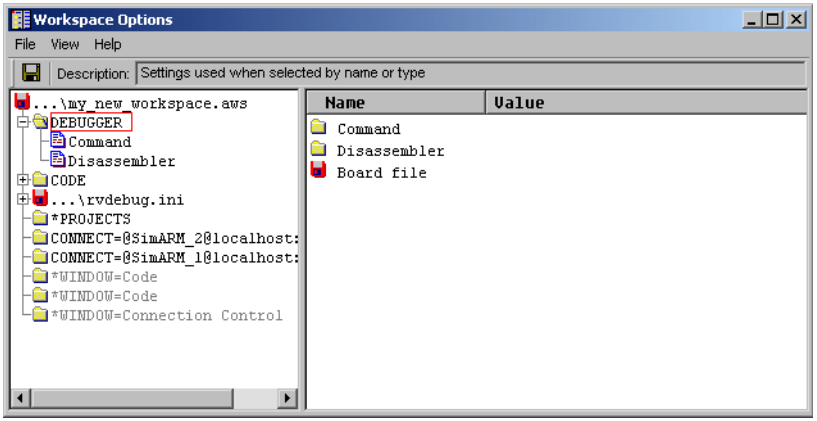

Figure 10-3 Example Workspace Options

If you click on an entry in the left pane, a red box is drawn around it and the Description field is updated. At the same time, the right pane, the Settings Values pane, is updated to show the contents of the highlighted group (see *Settings Values pane*).

Settings Values pane

If you click on an entry in the left pane, a red box is drawn around it and the Description field is updated. At the same time, the right pane, the Settings Values pane, is updated to show the contents of the highlighted group. Groups of settings are associated with an icon to explain their function:

Red disk This specifies a disk file.

RealView Debugger uses this, for example, to specify a board file.

Yellow folder

This is a parent group or rules page containing other groups (*rules pages*) and/or entries.

String value This is a text string. If more than one value can be assigned to the setting, a new setting is created with the chosen value, and is colored blue. The original setting remains available for you to add other values if required.

Numerical value

This is a numerical value.

True/False value

This has a value that is either `True` or `False`. To change the value, either:

- click on the switch button in the Value field
- right-click on the setting, and select `True` or `False` from the context menu.

Preset selections value

This enables you to select the value from a list that is defined by RealView Debugger. If more than one value can be assigned to the setting, a new setting is created with the chosen value, and is colored blue. The original setting remains available for you to add other values if required.

An asterisk (*) is placed at the front of a setting to show that it has changed from the default or that it was changed by RealView Debugger.

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:

- *Restoring settings*
- *Changing settings* on page 10-16
- *Copying entries* on page 10-17
- *Pasting entries* on page 10-17
- *Cutting entries* on page 10-19
- *Resetting entries* on page 10-19.

10.3.1 Restoring settings

If you have made changes to your workspace settings file, or your global configuration file, and you want to restore the factory settings:

1. Exit RealView Debugger in the normal way.
2. Locate the `.aws` and the `.ini` files in your RealView Debugger home directory.
3. Delete both files.
4. Start RealView Debugger and accept the option to create a new `.aws` file.

10.3.2 Changing settings

This section includes examples of changes that you can make to your current workspace. Select **File → Workspace → Workspace Options...** to display the Workspace Options window to edit the workspace file. This means that changes take effect in the current workspace:

- *Configuring the Code window*
- *Setting up debugger options*.

When you have saved a change, select **View → New Code Window** to display a new Code window to see the effect.

Configuring the Code window

To change the size of the Code window:

1. Expand the `CODE` group.
2. Expand the `Pos_size` group.
3. Right-click on the default `Num_lines` setting.
4. Select **Edit Value** from the context menu.
5. Use in-place editing to set the value to `0x040`.
6. Press Enter to confirm your setting.
7. Save the updated version of the workspace settings file.

Note

To restore the Code window, select the option **Reset to Empty**.

Setting up debugger options

To change the height of the Output pane:

1. Expand the `DEBUGGER` group.
2. Expand the `Command` group.
3. Right-click on the default `Num_lines` setting.
4. Select **Edit Value** from the context menu.
5. Use in-place editing to set the value to `10`.

6. Press Enter to confirm your setting.
7. Save the updated version of the workspace settings file.

Note

To restore the Code window, select the option **Reset to Empty**.

10.3.3 Copying entries

When you are working in the Workspace Options window, context menus include options to enable you to copy settings so that you can make changes quickly. The options that are available depend on the:

- pane you are working in
- contents of the clipboard
- relationship between what is on the clipboard and the entry under the cursor when you right-click.

The available options are:

Copy Select this option to make a copy of an entry to the clipboard ready for pasting.

This option is always available in the left pane to copy settings groups. When you are working in the right pane, this option depends on the entry under the cursor.

Make Copy...

Where permitted, this option enables you to make a copy of the chosen group. A dialog box enables you to define a new name for the copy.

10.3.4 Pasting entries

When you are working in the Workspace Options window, context menus include options to enable you to paste settings so that you can make changes quickly. Like the copy options, the paste options that are available depend on the:

- pane you are working in
- contents of the clipboard
- relationship between what is on the clipboard and the entry under the cursor when you right-click.

The available options are:

Paste Group Into

This option is only available if you right-click on a settings group, or a container disk file, with a settings group already copied to the clipboard.

This option is usually available in the left pane to paste settings groups into the settings file, or to copy between the workspace settings file and the global configuration file.

When you are working in the right pane, this option depends on the entry under the cursor. This means that you might not be able to paste the contents of the clipboard into the chosen location.

Paste Rule Here

Certain settings in the right pane are classed as *rules*. In particular, you can use rules to specify settings for projects when you are working in the Project Properties window. See the chapter that describes customizing projects in *RealView Debugger v1.8 Project Management User Guide* for details.

This option is only available if you right-click on a settings group, or a container disk file, with a single rule setting already copied to the clipboard.

This option is available in the left or right pane if you select a settings group that can accept the rule currently on the clipboard. This means that you might not be able to paste the contents of the clipboard into the chosen location.

Paste as 1st Child

To see this option, you must right-click on a parent group with a child group already copied to the clipboard.

Use this option, in the left pane, to paste a settings group into a parent group, that is to create a sibling group.

When you are working in the right pane, use this option to paste a child group. The paste only succeeds if the chosen parent group can accept the child. This means that you might not be able to paste the contents of the clipboard into the chosen location.

This option might be replaced by **Paste After**. This depends on the entry under the cursor.

Paste After Use this option, like **Paste as 1st Child**, to paste a settings group into a parent group, that is to create a sibling group.

This option enables you to specify the relationship between the new group and other siblings. This determines the order in which settings are used.

Paste String This option is only available in the right pane if you right-click on a settings group, or a container disk file, with a string setting already copied to the clipboard.

The paste only succeeds if you select a setting that can accept the string currently on the clipboard. This means that you might not be able to paste the contents of the clipboard into the chosen location.

Paste Value This option is only available in the right pane if you right-click on a settings group, or a container disk file, with a value setting already copied to the clipboard.

The paste only succeeds if you select a setting that can accept the value currently on the clipboard. This means that you might not be able to paste the contents of the clipboard into the chosen location.

10.3.5 Cutting entries

When you are working in the Workspace Options window, context menus include the option **Cut** to enable you to mark an entry for deletion. The entry is grayed out until you paste it into a new location.

If you cut another entry before you paste the first entry, the first entry is restored.

If you cut an entry, do not delete it until it has been pasted. If you delete the cut entry, it is no longer available to paste. This also applies to entries that you copy.

Always use a **Delete** option to remove unwanted entries.

10.3.6 Resetting entries

An asterisk is placed at the front of any entry that you edit in the workspace settings file to show that it has changed from the default. This also applies to entries maintained by RealView Debugger. You can reset all values using the **File → Reset** option from the window menu. This updates the window with the settings currently saved on disk. You are warned that any changes made since you last saved are lost.

When you have changed a value, right-click to see the context menu showing the option **Reset to Default**. Select this to change the value to the default and cancel any changes.

Right-click on a group of settings and select **Delete Contents** from the context menu to reset it back to empty. This deletes all the changed settings and restores the defaults. There is no undo.

Appendix A
Workspace Settings Reference

This appendix contains reference details about settings that define the RealView® Debugger workspace and global configuration options. It contains the following sections:

A.1 DEBUGGER

Settings in this group govern the behavior of generic actions in the debugger. These controls are then used in conjunction with other processor-specific controls.

DEBUGGER contains two second-level groups and a file:

- *Command*
- *Disassembler* on page A-3
- *Board_file* on page A-5.

A.1.1 Command

Settings in this group control the behavior and appearance of the Code window command line and Output pane. Use these to customize the input and output format used in this area.

When RealView Debugger starts, it uses the last-used settings unless overridden by settings in this group. These settings can be overridden dynamically by issuing CLI commands.

Saving changes takes immediate effect or at next start-up.

Table A-1 describes the settings.

Table A-1 Command settings

Name	Properties
Num_lines	The height of the Output pane. The default setting is 5 lines. Values can be entered in hex or decimal, for example 15 or 0x000F.
Radix_in	This setting specifies the format of number input options at start-up. The default format is decimal. However, you can enter hex numbers, for example ce number=0xABCD. Switching to hex also enables you to enter decimal numbers, for example ce number=01234t.
Radix_out	This setting specifies number format output options at start-up. The default format is decimal.

Table A-1 Command settings (continued)

Name	Properties
Char	The way that char* and char[] values are displayed, for example as strings or values, for the PRINT command.
Uchar	The way that unsigned char* values are displayed, for example as strings or values, for the PRINT command.
Buffer_height	The number of lines of scrollback available in the Command area. The default is 1024 lines. Values must be greater than 32. Changing this takes effect when RealView Debugger next starts.

A.1.2 Disassembler

Settings in this group control how the disassembly view is displayed in the Code window. This can be set for all processors or for specific processors only. The default settings apply to all processors.

When RealView Debugger starts, it uses the last-used settings unless overridden by these settings. These settings can be overridden dynamically by issuing CLI commands.

Saving changes takes immediate effect.

Note

Some processor disassemblers do not support features configured with these settings, and the settings are ignored.

Table A-2 describes the settings.

Table A-2 Disassembler settings

Name	Properties
Symbols	When instructions reference direct memory locations, either relative to the PC or as absolute references, the debugger tries to show the symbol at that location. Use this to disable this property.
Labels	When an instruction has a label associated with the address, the debugger shows it inline. Use this to disable this property.
Source	By default, high-level source code is interleaved with disassembly code when available. Use this to disable this property.

Table A-2 Disassembler settings (continued)

Name	Properties
Asm_source	By default, assembler source code is interleaved with disassembly code when available. This is isolated from high-level language source code display because the assembly source is usually of less interest in this mode. Not all assemblers produce the information to enable assembly tagging. This setting does not affect what is shown in the **Src** tab in the File Editor pane.
Source_line_cnt	By default, interleaved display shows eight lines of source code for any instruction. If there are more source lines associated with the instruction, they are not shown. Use this to define how many lines are shown, or to specify that all are shown.
Stack_syms	Some disassemblers identify frame or stack offset references in the operand fields. Use this to display the corresponding stack-based variable when possible.
Register_syms	Not available with ARM® tools. Some disassemblers identify register usage in the operand fields. Use this to show the corresponding register-based variable when possible.
Format	Some disassemblers include alternate format which is processor-specific. By default, RealView Debugger shows the format that is most appropriate for the given processor context. Use this to change the default format. You can also change the format dynamically using the DISASSEMBLE command, that is DISASSEMBLE /D, for default format, or DISASSEMBLE /A, for alternate formatWhen debugging ARM processors, use this to force the disassembler to display 16-bit values, as opposed to showing the appropriate format for the given processor context, that is mixed 16-bit and 32-bit values. Some DSPs use this to differentiate mnemonic and algebraic format.
Instr_value	In general, disassemblers show instructions as values and as opcodes or operands. Use this to suppress the value display where possible.

A.1.3 Board_file

Change this setting to specify a different board file for the current session.

Changing this value takes immediate effect. The specified board file is read and the contents used to populate the configuration details.

Resetting the value back to `Empty` does not take effect until the next time RealView Debugger starts.

A.2 CODE

Settings in this group govern the behavior of all Code windows when running a debugging session. These settings control the display characteristics of windows, their size and position, and any user-defined buttons created on the toolbars (not available in this release).

CODE contains two second-level groups and a settings rule:

- *Pos_size*
- *Button*
- *Asm_type* on page A-7.

A.2.1 Pos_size

Settings in this group control the position on screen and the size of Code windows in lines and characters. Use these to customize the size and position of Code windows in the debugger.

On start-up, RealView Debugger uses the last-used settings unless overridden by these settings. As you open new Code windows in the session, they are controlled by these settings.

Table A-3 describes the settings.

Table A-3 Pos_size settings

Name	Properties
Num_lines	Use this to specify window height as a given number of lines. The default is 0x0200.
Num_chars	Use this to specify window width as a given number of characters. The default is 0x02A0.
X_pos	Use this to specify the X position of the top-left corner of the window. The default is 0x010F.
Y_pos	Use this to specify the Y position of the top-left corner of the window. The default is 0x0066.

A.2.2 Button

Note

These settings are not available in this release.

A.2.3 Asm_type

When an assembler source file opens, RealView Debugger decides what type of processor is in use. However, the processor type is unknown if:

- there are no active connections
- there are no user-defined projects currently open
- there are no auto-projects currently open.

In this case, RealView Debugger does not know the format of instructions and so cannot define source coloring rules. This generates a selection box where you can specify the processor type.

Change this to specify a default processor type on start-up, for example `ARM`.

Saving a change to this setting takes immediate effect on new assembler source files opened following the update.

A.3 ALL

This group contains three second-level groups:

- *Text*
- *Search* on page A-11
- *Edit* on page A-12.

A.3.1 Text

Settings in this group control the File Editor pane and editor functions within the Code window.

Saving changes might not take effect until the next time RealView Debugger starts, or when a new Code window opens.

The `Text` group contains third-level groups and a series of settings:

Height Use this setting to specify the height, in number of lines, for text displayed in the File Editor pane.

Width Use this setting to specify the width, in number of characters, for text displayed in the File Editor pane.

Src_color_dis

Source coloring is used to make it easier to read source of high-level and low-level languages. All source coloring can be disabled in which case all text is the same color (usually black).

By default, source coloring is enabled, that is this setting is `False`.

Internationalization

Settings in this group configure multiple language support.

Table A-4 describes these settings.

Table A-4 Internationalization settings

Name	Properties
`Enabled`	Use this to enable or disable internationalization. By default, internationalization is disabled, that is this setting is `False`.
`Language`	Use this to specify the language to use for text. The options are: • `English` (default) • `Japanese` • `Chinese`. For Chinese and Japanese languages, see the font recommendations in the *RealView Debugger v1.8 Essentials Guide*.
`Default_encoding`	Use this to specify the default text encoding. The options are: • `ASCII` (default) • `UTF-8` • `Locale`.

Font_information

This group contains the `Pane_font` setting to set the font used in panes:

- On Windows, a Font dialog box is displayed to enable you to change the font settings shown in Table A-5. This table also shows the default font settings.

Table A-5 Default font settings (Windows)

Setting	Default Value
Font	`Courier New`
Font style	`Regular`
Size	`9`
Script	`Western`

- On Sun Solaris and Red Hat Linux, a dialog box is displayed where you must type in the font information directly. The default font is `Courier New`. The font information must be in the following format:

  `font_name,n,n,n,n,n,n,n,n,n,n,n,n`

  For example, to set the equivalent font settings to those given in Table A-5, enter:

```
Courier New,-12,0,0,0,400,0,0,0,0,3,2,1,49
```

Source_coloring

These settings control the colors used to identify source tokens. The defaults have been chosen to be easy to read and work well to isolate different program areas. The coloring choices are made relative to the built-in color models.

If you attempt to set the same foreground and background color, then RealView Debugger uses the foreground color but uses the default color for the background.

Table A-6 describes the settings.

Table A-6 Source_coloring settings

Name	Properties
`File_extensions`	The standard C/C++ source coloring is auto-enabled based on file extension. Use this to specify a comma-separated list of file extensions that, when loaded, trigger source code coloring.
`Scheme`	The color scheme you want to use for source coloring. You can select one of: • `Default` for the RealView Debugger v1.8 coloring scheme • `VisualStudio` for the Visual Studio coloring scheme • `CodeWarrior` for the CodeWarrior coloring scheme • `RVD.1.7` for the RealView Debugger v1.7 coloring scheme.
`Numbers_text`	Use this to specify the foreground color for numbers displayed in the File Editor pane.
`Numbers_bkgrnd`	Use this to specify the background color for numbers displayed in the File Editor pane.
`Strings_text`	Use this to specify the foreground color for strings displayed in the File Editor pane.
`Strings_bkgrnd`	Use this to specify the background color for strings displayed in the File Editor pane.
`Keywords_text`	Use this to specify the foreground color for C/C++ keywords displayed in the File Editor pane.
`Keywords_bkgrnd`	Use this to specify the background color for C/C++ keywords displayed in the File Editor pane.
`Comments_text`	Use this to specify the foreground color for comments displayed in the File Editor pane.

Table A-6 Source_coloring settings (continued)

Name	Properties
Comments_bkgrnd	Use this to specify the background color for comments displayed in the File Editor pane.
Identifiers_text	Use this to specify the foreground color for identifiers displayed in the File Editor pane.
Identifiers_bkgrnd	Use this to specify the background color for identifiers displayed in the File Editor pane.
User_text	Use this to specify the foreground color for user-defined keywords displayed in the File Editor pane.
User_bkgrnd	Use this to specify the background color for user-defined keywords displayed in the File Editor pane.
Preprocessor_text	Use this to specify the foreground color for preprocessor keywords (#keyword) displayed in the File Editor pane.
Preprocessor_bkgrnd	Use this to specify the background color for preprocessor keywords (#keyword) displayed in the File Editor pane.
Operator_text	Use this to specify the foreground color for operators displayed in the File Editor pane.
Operator_bkgrnd	Use this to specify the background color for operators displayed in the File Editor pane.
User_keywords	Use this to specify a list of user-defined keywords that are highlighted when they appear in the File Editor pane.

A.3.2 Search

Settings in this group control the searching behavior when working with source files in the File Editor pane.

These settings can be overridden dynamically using the menus and toggles in the File Editor.

Table A-7 describes these settings.

Table A-7 Search settings

Name	Properties
`Direction`	Use this to specify the search direction. The default is to search forwards, that is, from the top to the bottom of the file.
`Wrap`	Use this to specify search behavior when the end of file is reached. The default is to wrap during a search, that is, to search to the end of the file and then to start again at the top until the starting point is reached.
`Sensitive`	Use this to specify whether uppercase and lowercase characters are treated as identical in searches. By default, searches are case-sensitive.
`Regexp`	When set to `True`, full grep-style regular expressions are used in searches. The default is `False`, not enabled.
`Fail`	Use this to specify editor behavior when a search fails. Set to `dialog` by default, you can change this to `flash`.

A.3.3 Edit

Settings in this group control general editor behavior when working with source files in the File Editor pane.

These settings can be overridden dynamically using the menus and toggles in the File Editor.

The `Edit` group contains three third-level groups and a series of settings:

Backup

These settings control the backup behavior when working with source files in the File Editor pane.

Table A-8 describes these settings.

Table A-8 Backup settings

Name	Properties
`Disable`	By default, a backup file is created when a file is edited. This provides a useful safety feature. Use this to disable this feature if required.
`Backup_dir`	By default, backup files are saved in the same directory as the original file. Use this to specify a pathname to a new location, for example to keep all backup files in one special directory.
`Backup_ext`	By default, backup files are saved with the `.bak` extension appended to the original filename.

Tab_conv

Settings in this group control the display behavior when working with source files in the File Editor pane. These settings are used to handle tabs and spaces.

Tabs are permitted in files and are left untouched, by default. Use these settings to convert tabs to spaces when writing to the file, that is saving, and to convert spaces to tabs when reading the file.

Spaces are not converted to tabs inside “ and “ quoting blocks on a line.

Table A-9 describes these settings.

Table A-9 Tab_conv settings

Name	Properties
`Tabs_to_spaces`	Converts tabs to spaces when the file is saved.
`Spaces_to_tabs`	Converts spaces to tabs when the file is read.
`To_spaces_ext`	Use this to specify file extensions where tab conversions take place. Specify a list separated by semi-colons (;)
`To_tabs_ext`	Use this to specify file extensions where space conversions take place. Specify a list separated by semi-colons (;).

Src_ctrl

Settings in this group control source control access tools when working in the File Editor pane and in the debugger. The editor attempts to detect any source control system in use, where possible, but you might have to specify it before RealView Debugger can use it.

Use these settings to specify complete source control commands, and to override commands for known systems.

The `Src_ctrl` group contains a low-level group and a series of settings:

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.

For full details on these settings see the chapter that describes version control in *RealView Debugger v1.8 Project Management User Guide*.

Edit

Settings in this group configure editor behavior when working with source files in the File Editor pane.

Table A-10 describes these settings.

Table A-10 Edit settings

Name	Properties
`Drag_drop_dis`	Use this to disable drag-and-drop editing when working in the File Editor pane.
`Vi`	Not used.
`Indent`	Use this to set indenting so that a specified number of spaces are inserted as you open a new line. By default, auto-indent inserts the same number of spaces as on the previous line. If the previous line is a left curly bracket (`{`) the shift is increased. If the previous line is a right curly bracket (`}`), shift spaces are subtracted.
`Undo`	Use this to specify the levels of undo and redo. By default, this is set to `64`.
`Tab`	Use this to specify the size of TAB settings when working in the File Editor. By default, this is set to `8`. Use a value between `1` and `16`.
`Shift`	Use this to specify the size of shift spaces as used in the Indent rule and accessed through the Code window Edit menu options. By default, this is set to `2`. Use a value between `2` and `32`.
`Line_number`	By default, line numbering is disabled in the debugger and the File Editor. Use this to change the editor default to show line numbers at start-up.

Table A-10 Edit settings (continued)

Name	Properties
No_tooltip	By default, tooltip evaluation of variables and registers is enabled. Change this setting to True to disable this feature.
Timer	During file editing, the editor periodically checks to see if another tool has edited or deleted the files being tested. A warning is shown if an update is detected. Use this to specify the number of seconds between checks. The default is 60 seconds. Use values greater than 30 seconds. Set to -1 to disable this feature.
Tool_save	When performing a build, you are prompted to resave any files that have been edited. Use this to specify automatic resaving of changed files at build time to ensure that your latest sources are included. You can also set a no-save, no-ask value.
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	Not used.

Appendix B
RealView Debugger on Sun Solaris and Red Hat Linux

This appendix provides supplementary information for developers using RealView® Debugger on Sun Solaris and Red Hat Linux. It contains the following sections:

B.1 About this Appendix

This appendix describes features that are specific to using RealView Debugger v1.8 on Sun Solaris and Red Hat Linux, and contains corrections and additions to the documentation suite. It covers:

- *RealView Debugger Essentials Guide*
- *RealView Debugger User Guide*
- *RealView Debugger Project Management User Guide*
- *RealView Debugger Target Configuration Guide*
- *RealView Debugger Command Line Reference Guide*
- *RealView Debugger Extensions User Guide*.

B.2 Getting more information

This section describes how to get more information on RealView Debugger:

- *Online resources*
- *Feedback on RealView Debugger*.

B.2.1 Online resources

ARM® Limited 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
- product Release Notes
- updates to documentation
- Frequently Asked Questions.

You can access these resources from the ARM web site at `http://www.arm.com`.

B.2.2 Feedback on RealView Debugger

If you have any problems with RealView Debugger, submit a *Software Problem Report* (SPR):

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.

B.3 X-Windows configuration change required on Sun Solaris

If you are using RealView Debugger on Sun Solaris, you must modify the X-Windows configuration file (`.Xdefaults`) located in your `$HOME` directory. The file must contain the following line:

```
Dtwm*secondariesOnTop: True
```

The case is important so enter the line exactly as shown. If the `.Xdefaults` file does not exist then you must create it.

If you do not do this, then when you change focus from a floating pane to the Code window, the floating pane is hidden behind the Code window.

B.4 Changes to target configuration details

This section describes additions to *RealView Debugger v1.8 Target Configuration Guide:*

- *Default configuration files*
- *Target configuration entries*.

B.4.1 Default configuration files

Because you are using RealView Connection Broker connections to *RealView ARMulator® ISS* (RVISS), the RDI configuration files, described in the documentation suite, are not installed when running RealView Debugger on Sun Solaris or Red Hat Linux, for example `*.rbe` or `*.cnf` files. Other configuration files are installed in:

`install_directory``/RVD/Core/.../solaris-sparc/etc` (Sun Solaris)

`install_directory``/RVD/Core/.../linux-pentium/etc` (Red Hat Linux)

These files are:

- 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 that describes configuring custom targets in *RealView Debugger v1.8 Target Configuration Guide.*

B.4.2 Target configuration entries

RealView Connection Broker enables you to connect to local or remote simulators, such as RVISS. You can also connect to local or remote *On-Chip Debugging* (OCD) based emulators such as Multi-ICE® direct connect.

This means that target configuration entries related to remote connections are fully supported by RealView Debugger, for example the `Remote` group, shown in Figure B-1 on page B-6.

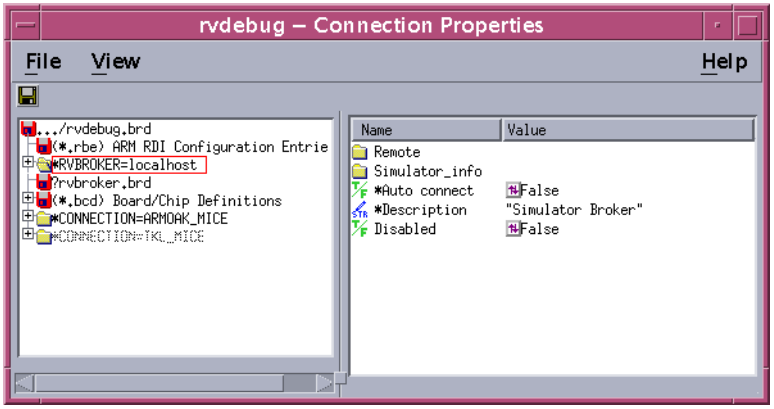

Figure B-1 Remote entries in the Connection Properties window

See the chapter that describes working with remote targets in *RealView Debugger v1.8 Target Configuration Guide* for details on how to make connections using these settings.

Using the DSP

The DSP support in RealView Debugger is invoked by connecting the debugger to a DSP processor. Developers working on Sun Solaris or Red Hat Linux can connect to a simulated target on a remote workstation or connect to target hardware using Multi-ICE direct connect or RealView ICE. See the chapter that describes working with remote targets in *RealView Debugger v1.8 Target Configuration Guide* for details.

Note

RealView Debugger DSP support is a separately licensed component. See the chapter that describes DSP support in *RealView Debugger v1.8 Extensions User Guide* for full details on using this extension.

B.5 Changes to GUI and general user information

This section describes:

- *Getting started with RealView Debugger*
- *Column resizing* on page B-8
- *Saving favorites* on page B-10
- *Changes to the desktop* on page B-10
- *RealView Debugger examples* on page B-11.

B.5.1 Getting started with RealView Debugger

The RealView Debugger *Control Panel*, `RVDEBUGCP`, provides access to the debugger for developers using Sun Solaris and Red Hat Linux. Start the debugger to see this window, shown in Figure B-2.

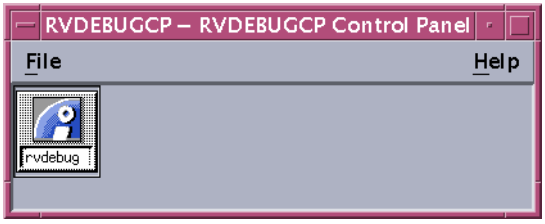

Figure B-2 RealView Debugger Control Panel

By default, launching the Control Panel starts RealView Debugger without a splash screen.

Click on the **rvdebug** icon to access startup options for RealView Debugger:

- Double-click on this icon to start the debugger in default mode and display the Code window.
- Left-click on this 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.

You can also start RealView Debugger from the command line with the `rvdebug` command. See *Starting from the command line* on page 1-2 for details of startup options and command-line arguments.

Note

If you are using Sun Solaris or Red Hat Linux, you can use the `-cmd` argument to start the command-line debugger, for example `rvdebug -cmd`.

B.5.2 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 Sun Solaris and Red Hat Linux:

1. Select **Target → Reload Image to Target** to reload the image `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Set a default breakpoint by double-clicking on line 183.
4. Click **Run** to start execution.
5. Enter `5000` when asked for the number of runs.

   The program starts and then stops when execution reaches the breakpoint at line 183. The red box marks the location of the *Program Counter* (PC) when execution stops.
6. If you do not have a Watch pane visible, select **View → Watch** to display the Watch pane.
7. Right-click somewhere in the header columns to display the context menu, shown in Figure B-3 on page B-9.

Figure B-3 Column resizing in the Watch pane

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 B-4.

Figure B-4 Column resizing prompt box

   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.

B.5.3 Saving favorites

Be aware of the following when working on Sun Solaris or Red Hat Linux:

- The history file, `exphist.sav`, is only created if you create and save a favorite, for example a breakpoint or watchpoint.
- The history file is not controlled in the same way when you access an **Open...** dialog.

When the file has been created and saved, it is updated at the end of each debugging session in the usual way.

B.5.4 Changes to the desktop

This section summarizes differences when using the RealView Debugger desktop on Sun Solaris or Red Hat Linux.

Code window

Be aware of the following when working in the 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. However, the target processor is not shown.
- The Code window includes a Color Box to identify the connection associated with the window. Other windows, such as the Resource Viewer, do not include a Color Box to identify the calling Code window.
- Tooltips are not available if you hover the mouse cursor over a toolbar button. However, a button description is displayed in the Status line.
- The pane name is not displayed if you hover the mouse cursor over the pane Reposition control.
- The editing control called **Tooltip Evaluation** that provides hover-style evaluation in different code views is not available.
- Select **Help** → **About...** from the RealView Debugger main menu to display the About Box Information dialog. This also enables you to submit a Software Problem Report.

- Developers working on Red Hat Linux must use the keyboard shortcut Ctrl+Shift+F10 to step over functions (see *Using shortcuts* on page 3-9).

B.5.5 RealView Debugger examples

Be aware of the following when working on Sun Solaris or Red Hat Linux.

Depending on your installation, the RealView Debugger examples for developing applications on ARM targets, are installed in the examples directory in:

install_directory/RVDS/Examples/...

The dhrystone example project, and the executable built by this project, are used in the documentation suite.

See the tutorial in *RealView Debugger v1.8 Essentials Guide* for more details on building this example project.

This location is also used, by default, if you choose a Custom installation and specify that the RealView Debugger examples are installed.

Tools support examples

If you choose a Custom installation to install RealView Debugger Tools support for CEVA-Oak and CEVA-TeakLite, DSP examples are installed in the examples directory:

install_directory/RVD/Tools/Oak-Teaklite/.../demo_DSPG

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 architecture-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. ADS is superseded by RealView Developer Suite (RVDS).

See also RealView Developer Suite.

ARM instruction A word that encodes an operation for an ARM processor operating in ARM state. ARM instructions must be word-aligned.

ARM state A processor that is executing ARM instructions is operating in ARM state. The processor switches to Thumb state (and to recognizing Thumb instructions) when directed to do so by a state-changing instruction such as `BX`, `BLX`.

See also Thumb state.

Backtracing *See* Call Stack.

Big-endian Memory organization where the least significant byte of a word is at the highest address and the most significant byte is at the lowest address in the word.

See also Little-endian.

Board RealView Debugger uses the term *board* to refer to a target processor, memory, peripherals, and debugger connection method.

Board file The *board file* is the top-level configuration file, normally called `rvdebug.brd`, that references one or more other files.

Breakpoint A user defined point at which execution stops in order that a debugger can examine the state of memory and registers.

See also Conditional breakpoint, Default breakpoint, Hardware breakpoint, Software breakpoint, and Unconditional breakpoint.

Call Stack This is a list of procedure or function call instances on the current program stack. It might also include information about call parameters and local variables for each instance.

Conditional breakpoint
A *breakpoint* that has a condition qualifier assigned, and so halts execution when that condition becomes True. The condition normally references the values of program variables that are in scope at the breakpoint location.

See also Breakpoint, Default breakpoint, Hardware breakpoint, Software breakpoint, and Unconditional breakpoint.

Context menu *See* Pop-up menu.

Core module In the context of Integrator, an add-on development board that contains an ARM processor and local memory. Core modules can run stand-alone, or can be stacked onto Integrator motherboards.

See also Integrator.

Current Program Status Register (CPSR)
See Program Status Register.

DCC *See* Debug Communications Channel.

Debug Agent (DA) The Debug Agent resides on the target to provide target-side support for *Running System Debug* (RSD). The Debug Agent can be a thread or built into the RTOS. The Debug Agent and RealView Debugger communicate with each other using the *debug communications channel* (DCC). This enables data to be passed between the debugger and the target using the ICE interface, without stopping the program or entering debug state.

See also Running System Debug.

Debug Communications Channel (DCC)
A debug communications channel enables data to be passed between RealView Debugger and the EmbeddedICE logic on the target using the JTAG interface, without stopping the program flow or entering debug state.

Debug With Arbitrary Record Format (DWARF)
ARM code generation tools generate debug information in DWARF2 format by default. From RVCT v2.2, you can optionally generate DWARF3 format (Draft Standard 9).

Default breakpoint
A *breakpoint* that does not have a condition qualifier or an action assigned. A default breakpoint is always an *unconditional breakpoint*.

See also Breakpoint, Conditional breakpoint, Hardware breakpoint, Software breakpoint, and Unconditional breakpoint.

Deprecated A deprecated option or feature is one that you are strongly discouraged from using. Deprecated options and features are to be removed in future versions of the product.

Digital Signal Processor (DSP)
DSPs are special processors designed to execute repetitive, maths-intensive algorithms. Embedded applications might use both ARM processor cores and DSPs.

Doubleword A 64-bit unit of information.

DSP *See* Digital Signal Processor.

DWARF *See* Debug With Arbitrary Record Format.

ELF Executable and Linking Format. ARM code generation tools produce objects and executable images in ELF format.

Embedded Trace Macrocell (ETM)
A block of logic, embedded in the hardware, that is connected to the address, data, and status signals of the processor. It broadcasts branch addresses, and data and status information in a compressed protocol through the trace port. It contains the resources used to trigger and filter the trace output.

EmbeddedICE logic
The EmbeddedICE logic is an on-chip logic block that provides TAP-based debug support for ARM processor cores. It is accessed through the TAP controller on the ARM core using the JTAG interface.

See also IEEE1149.1.

Emulator In the context of target connection hardware, an emulator provides an interface to the pins of a real core (emulating the pins to the external world) and enables you to control or manipulate signals on those pins.

Endpoint connection
A debug target processor, normally accessed through an *access-provider connection*.

ETM *See* Embedded Trace Macrocell.

ETV *See* Extended Target Visibility.

Extended Target Visibility (ETV)
Extended Target Visibility enables RealView Debugger to access features of the underlying target, such as chip-level details provided by the hardware manufacturer or SoC designer.

Floating Point Emulator (FPE)
Software that emulates the action of a hardware unit dedicated to performing arithmetic operations on floating-point values.

FPE *See* Floating Point Emulator.

Halfword A 16-bit unit of information.

Halted System Debug (HSD)
Usually used for RTOS aware debugging, *Halted System Debug* (HSD) means that you can only debug a target when it is not running. This means that you must stop your debug target before carrying out any analysis of your system. With the target stopped, the debugger presents RTOS information to you by reading and interpreting target memory.

See also Running System Debug.

Hardware breakpoint

A breakpoint that is implemented using non-intrusive additional hardware. Hardware breakpoints are the only method of halting execution when the location is in *Read Only Memory* (ROM). Using a hardware breakpoint often results in the processor halting completely. This is usually undesirable for a real-time system.

See also Breakpoint, Conditional breakpoint, Default breakpoint, Software breakpoint, and Unconditional breakpoint.

HSD *See* Halted System Debug.

IEEE 1149.1 The IEEE Standard that defines TAP. Commonly (but incorrectly) referred to as JTAG.

See also Test Access Port

Integrator A range of ARM hardware development platforms. *Core modules* are available that contain the processor and local memory.

Joint Test Action Group (JTAG)

An IEEE group focussed on silicon chip testing methods. Many debug and programming tools use a *Joint Test Action Group* (JTAG) interface port to communicate with processors. For more information see IEEE Standard, Test Access Port and Boundary-Scan Architecture specification 1149.1 (JTAG).

JTAG *See* Joint Test Action Group.

JTAG interface unit A protocol converter that converts low-level commands from RealView Debugger into JTAG signals to the processor, for example to the EmbeddedICE logic and the ETM.

Little-endian Memory organization where the least significant byte of a word is at the lowest address and the most significant byte is at the highest address of the word.

See also Big-endian.

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 ARMulator ISS (RVISS)
The most recent version of the ARM simulator, RealView ARMulator ISS is supplied with RealView Developer Suite. It communicates with a debug target using RV-msg, through the RealView Connection Broker interface, and RDI.

See also RDI and RealView Connection Broker.

RealView Compilation Tools (RVCT)
RealView Compilation Tools is a suite of tools, together with supporting documentation and examples, that enables you to write and build applications for the ARM family of *RISC* processors.

RealView Connection Broker
RealView Connection Broker is an execution vehicle that enables you to connect to simulator targets on your local system, or on a remote system. It also enables you to make multiple connections to the simulator.

See also RealView ARMulator ISS.

RealView Debugger Trace
Part of the RealView Debugger product that extends the debugging capability with the addition of real-time program and data tracing. It is available from the Code window.

RealView ICE (RVI) A JTAG-based debug solution to debug software running on ARM processors.

Remote Debug Interface (RDI)
The *Remote Debug Interface* (RDI) is an ARM standard procedural interface between a debugger and the debug agent. RDI gives the debugger a uniform way to communicate with:

- a simulator running on the host (for example, RVISS)
- a debug monitor running on hardware that is based on an ARM core accessed through a communication link (for example, Angel)
- a debug agent controlling an ARM processor through hardware debug support (for example, RealView ICE or Multi-ICE).

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.

RSD *See* Running System Debug.

RTOS Real Time Operating System.

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

See also Debug Agent and Halted System Debug.

RVCT *See* RealView Compilation Tools.

RVISS *See* RealView ARMulator ISS.

Scan chain A scan chain is made up of serially-connected devices that implement boundary-scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain. Processors might contain several shift registers to enable you to access selected parts of the device.

Scope The range within which it is valid to access such items as a variable or a function.

Script A file specifying a sequence of debugger commands that you can submit to the command-line interface using the `include` command.

Semihosting A mechanism whereby I/O requests made in the application code are communicated to the host system, rather than being executed on the target.

Simulator A simulator executes non-native instructions in software (simulating a core).

Software breakpoint A *breakpoint* that is implemented by replacing an instruction in memory with one that causes the processor to take exceptional action. Because instruction memory must be altered software breakpoints cannot be used where instructions are stored in read-only memory. Using software breakpoints can enable interrupt processing to continue during the breakpoint, making them more suitable for use in real-time systems.

See also Breakpoint, Conditional breakpoint, Default breakpoint, Hardware breakpoint, and Unconditional breakpoint.

Software Interrupt (SWI)
An instruction that causes the processor to call a programmer-specified subroutine. Used by the ARM standard C library to handle semihosting.

SPSR Saved Program Status Register.

See also Program Status Register.

SWI *See* Software Interrupt.

Target The target hardware, including processor, memory, and peripherals, real or simulated, on which the target application is running.

Target vehicle Target vehicles provide RealView Debugger with a standard interface to disparate targets so that the debugger can connect easily to new target types without having to make changes to the debugger core software.

Target Vehicle Server (TVS)
Essentially the debugger itself, this contains the basic debugging functionality. TVS contains the run control, base multitasking support, much of the command handling, and target knowledge, such as memory mapping, lists, rule processing, board file and `.bcd` files, and data structures to track the target environment.

Thumb instruction One halfword or two halfwords that encode an operation for an ARM processor operating in Thumb state. Thumb instructions must be halfword-aligned.

Thumb state A processor that is executing Thumb instructions is operating in Thumb state. The processor switches to ARM state (and to recognizing ARM instructions) when directed to do so by a state-changing instruction such as `BX`, `BLX`.

See also ARM state.

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.

Unconditional breakpoint
: A *breakpoint* that does not have a conditional qualifier assigned, and so always halts execution.

  See also Breakpoint, Conditional breakpoint, Default breakpoint, Hardware breakpoint, and Software breakpoint.

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 watchpoints you have defined.

Watchpoint
: In RealView Debugger, this is a hardware breakpoint.

Word
: A 32-bit unit of information.

Index

C

G

H

I

J

L

M

N

O

P

Q

R

S

T