RealView® Developer Kit for Philips Nexperia Sy.Sol

Version 2.2

Getting Started Guide

ARM®

ARM DUI 0276B

RealView Developer Kit for Philips Nexperia Sy.Sol
Getting Started Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
July 2005	A	RVDK for Philips v2.2 release
January 2006	B	Includes trace support

Product Status

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

Web Address

`http://www.arm.com`

Contents
RealView Developer Kit for Philips Nexperia Sy.Sol Getting Started Guide

Preface

This preface introduces the *RealView® Developer Kit v2.2 for Philips Nexperia Sy.Sol*. It contains the following sections:

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

About this book

This book provides an overview of the *RealView® Developer Kit v2.2 for Philips Nexperia Sy.Sol* tools and documentation.

Intended audience

This book is written for all developers who are producing applications using *RealView Developer Kit* (RVDK). It assumes that you are an experienced software developer, but may not necessarily be familiar with RVDK and its component products.

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an introduction to RVDK v2.2.

Chapter 2 ***Features of the Compilation Tools, the Debugger, RealView ICE Micro Edition and RealView ICE***

Read this chapter for a description of the major features of the RVDK v2.2 compilation tools, debugger, RealView ICE Micro Edition, and RealView ICE.

Chapter 3 ***RealView Debugger Desktop***

Read this chapter for a detailed description of the contents of the RealView Debugger desktop.

Chapter 4 ***Getting Started with RealView Developer Kit***

Read this chapter for a high-level summary of the tasks involved in using the RVDK debugger and compilation tools.

Appendix A ***Using the armenv Tool***

Read this appendix for a description of how to manage your ARM RealView product installations.

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.
`monospace` (`mono` underlined)	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 ARM Limited that provide additional information on developing code for the ARM family of processors.

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

ARM Limited publications

This book contains general information about RVDK. Other publications included in the suite are:

- *RealView Developer Kit v2.2 Debugger User Guide* (ARM DUI 0281)
- *RealView Developer Kit v2.2 Compiler and Libraries Guide* (ARM DUI 0282)
- *RealView Developer Kit v2.2 Assembler Guide* (ARM DUI 0283)
- *RealView Developer Kit v2.2 Command Line Reference* (ARM DUI 0284)
- *RealView Developer Kit v2.2 Linker and Utilities Guide* (ARM DUI 0285)
- *RealView Developer Kit v2.2 Project Management Guide* (ARM DUI 0291)

- *RealView ICE and RealView Trace v1.5 User Guide* (ARM DUI 0155).
- *RealView ICE Micro Edition v1.1 User Guide* (ARM DUI 0220).

The following additional documentation is provided with RVDK:

- *RealView Developer Suite v2.2 CodeWarrior IDE Guide* (ARM DUI 0065)
- *ARM FLEXlm License Management Guide* (ARM DUI 0209).

Feedback

ARM Limited welcomes feedback on both this toolkit and its documentation.

Feedback on this toolkit

If you have any problems with *RealView Developer Kit v2.2 for Philips Nexperia Sy.Sol*, contact your supplier. To help them provide a rapid and useful response, give:

- your name and company
- the serial number of the product
- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- 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
- the version string of the tool, including the version number and date.

Feedback on this book

If you notice any errors or omissions in 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 the problem.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces *RealView® Developer Kit v2.2 for Philips Nexperia Sy.Sol* and describes the component products and documentation. It contains the following sections:

- *RealView Developer Kit components* on page 1-2
- *RealView Developer Kit licensing* on page 1-6
- *RealView Developer Kit documentation* on page 1-8.

1.1 RealView Developer Kit components

RVDK consists of a suite of tools, together with supporting documentation and examples, that enable you to write, build, and debug applications for your ARM architecture-based RISC processors.

RVDK supports the use of ARM926EJ-S™ and ARM946E-S™ processors, and the compilation of C and C++ (if licensed) source files. RVDK supports the debugging of single processor systems only. RVDK is supported on Windows 2000 and Windows XP Professional operating systems only.

Table 1-1 lists the components of RVDK v2.2, and the ARM product that is used to implement each component.

Table 1-1 RVDK component products

Component	ARM Product
Compilation tools	*RealView Compilation Tools v2.2*
Debugger	*RealView Debugger v1.8* on page 1-3
Debug agent	*RealView ICE Micro Edition v1.1* on page 1-4
Debug agent (optional)	*RealView ICE/RealView Trace v1.5* on page 1-4
Project manager	CodeWarrior® v5.6.1

The examples in the RVDK documentation do not refer specifically to your board details, but use a fictitious board to demonstrate the tasks you can perform (see *Names used in the debugger documentation examples* on page 1-5).

Also, the RVDK documentation refers to ARM architectures and processors that might not be supported with your version of RVDK.

Example source code and build files are provided with RVDK that demonstrate how to use the RVDK tools (see *RVDK example projects* on page 1-4 and *Using the example projects* on page 4-2).

1.1.1 RealView Compilation Tools v2.2

You can use *RealView Compilation Tools v2.2* to build C, C++ (if licensed), or ARM assembly language programs. See *Getting started with the compilation tools* on page 4-27 for an introduction on using the compilation tools.

RVCT short path names

The RVCT short path names shown in Table 1-2 are used throughout the RVCT documentation. The `install_directory` shown is the default installation directory. If you specified a different installation directory, then the path names are relative to your chosen directory.

Table 1-2 RVCT short path names

Short name	Default Directory
install_directory	`C:\Program Files\ARM`
program_directory	*install_directory*`\RVCT\Programs\`*2.2*`\`*build_num*`\philips\win_32-pentium`
includes_directory	*install_directory*`\RVCT\Data\2.2\`*build_num*`\include\windows`
examples_directory	*install_directory*`\RVDK\Examples\2.2\`*build_num*`\windows`

1.1.2 RealView Debugger v1.8

RealView Debugger v1.8, together with RealView ICE Micro Edition v1.1, enables you to debug your embedded application programs and have complete control over the flow of the program execution so that you can quickly isolate and correct errors. For a description of the basic tasks you can do with RealView Debugger, see Chapter 4 *Getting Started with RealView Developer Kit*.

RealView Debugger short path names

The RealView Debugger short path names shown in Table 1-3 are used throughout the debugger documentation. The `install_directory` shown is the default installation directory. If you specified a different installation directory, then the path names are relative to your chosen directory.

Table 1-3 RealView Debugger short path names

Short name	Default Directory
install_directory	`C:\Program Files\ARM`
program_directory	*install_directory*`\RVD\Core\1.8\`*build_num*`\philips\win_32-pentium`
examples_directory	*install_directory*`\RVDK\Examples\2.2\`*build_num*`\windows`

1.1.3 RealView ICE Micro Edition v1.1

RealView ICE Micro Edition v1.1, together with your target board and RealView Debugger v1.8, enables you to develop and debug your embedded applications.

1.1.4 RealView ICE/RealView Trace v1.5

This release includes support for trace functionality in RealView Debugger. The supplied RVI-ME unit does not support trace, so to make use of the trace functionality you must additionally purchase the RealView ICE (RVI) and RealView Trace (RVT) products.

RVI provides additional hardware and software to allow RVD to debug your device. The debug functionality of RVI is similar to RVI-ME, except that RVI can connect to your PC using Ethernet as well as USB, and RVI generally allows higher speed debugging (such as when downloading images). Even if you are connecting to your device using RVI, you must still have the RVI-ME unit that is supplied with this toolkit plugged into the PC, otherwise the tools will fail to operate.

RVT plugs-in to the RVI hardware and provides the capability to capture and store trace data from the trace port on your device. If your device contains an Embedded Trace Buffer (ETB) then you can capture trace directly using RVI, without using any RVT hardware. However, the quantity of trace that can be captured using an ETB is very small in comparison to RVT, so you may wish to use an RVT anyway.

To purchase RVI and RVT, contact your supplier or visit `http://www.arm.com`.

1.1.5 RVDK example projects

Example projects are provided with RVDK but have not been specifically modified for the target boards supported by this toolkit, and include:

- Projects that are located in the directory:

  install_directory\RVDK\Examples\2.2\build_num\windows

  You can open these example projects directly from the Windows **Start** menu:

  Programs → ARM → RealView Developer Kit v2.2 for Philips Nexperia Sy.Sol → Examples → ***project_name***

  ——— **Note** ———

  If you are using the default Windows XP settings, select **All Programs**.

  ————————————

  Whenever the **Examples** option is selected from the **Start** menu as described, a window displays to show the folders in which your example projects are located.

To build these projects, see *Building and rebuilding an image with RealView Debugger* on page 4-21.

1.1.6 Names used in the debugger documentation examples

Many of the examples in the debugger documentation do not refer to a specific board or debug target, but use the following names:

- `RVI` or `RVI-ME` used as an example access-provider connection
- `MCUeval` is used as an example board file name
- `ARM946E-S` is used as an example chip name
- `ARM946E-S_0` is used as an example debug target processor name.

When working through the examples, substitute the names that appear in your RealView Debugger interface.

1.2 RealView Developer Kit licensing

RVDK supports the following families of Philips Nexperia devices:

- `Sy.Sol 5200`
- `Sy.Sol 6100`
- `Sy.Sol 7200`

The components that comprise RVDK are licensed using the following mechanisms:

- RVI-ME dongle
- FLEX*lm* software license

The RVI-ME unit must be plugged into your PC and the FLEX*lm* license must be present in order for the RVDK components to be used. For further information on FLEX*lm*, refer to the *ARM FLEXlm License Management Guide*.

All versions of RVDK have the following limitations:

- The RVI-ME unit is compatible only with these tools, and cannot be used with any other ARM products.
- The RVI/RVI-ME must be connected to a board containing a permitted ARM processor. Each version of RVDK is designed to be used only with a particular processor, or range of processors, as detailed in *RealView Developer Kit components* on page 1-2.
- FLEX*lm* licenses for this RVDK are supplied only as node-locked licenses. This means you can only use RVDK on one specific system.

——— **Note** ———

Both the RVI-ME unit and the FLEX*lm* (software) licenses must be present in order for the debugger to operate.

The debugger will only work if the RVI/RVI-ME connection is to a processor that is permitted to be used with the toolkit.

————————

The available versions of this RVDK are described in Table 1-4.

Table 1-4 RVDK versions

RVDK version	License key times	C++ available as a license upgrade
RVDK v2.2 for Philips Nexperia Sy.Sol - Loaner Edition	45 days	-
RVDK v2.2 for Philips Nexperia Sy.Sol - Basic Edition	1 year (renewable)	✓
RVDK v2.2 for Philips Nexperia Sy.Sol - Developer Edition	Unlimited	✓

1.3 RealView Developer Kit documentation

See the *Further Reading* sections in each book for related publications from ARM, and from third parties.

1.3.1 Getting more information

The full RVDK documentation suite is available as PDF files.

To view the PDF files, select the following from the Windows **Start** menu:

Programs → ARM → RealView Developer Kit v2.2 for Philips Nexperia Sy.Sol → PDF Documentation

Note

If you are using the default Windows XP settings, select **All Programs**.

The PDF files are installed in the following directories:

- RealView Developer Kit v2.2 compilation tools and debugger documentation:

  `install_directory\Documentation\RVDK\2.2\release\windows\PDF`

- RealView ICE Micro Edition v1.1 documentation:

  `install_directory\Documentation\RVDK\2.2\release\windows\PDF`.

Rogue Wave C++ library manuals

If required, the manuals for the Rogue Wave C++ library are provided with RVDK in HTML files. To view these manuals, enter the following location in a web browser, such as Netscape Communicator or Internet Explorer:

`install_directory\Documentation\RogueWave\1.0\release\stdref\index.htm`

Chapter 2
Features of the Compilation Tools, the Debugger, RealView ICE Micro Edition and RealView ICE

This chapter describes the features of the RVDK compilation tools, the debugger, and the debug target hardware. It contains the following sections:

2.1 ARM Toolkit Proprietary ELF format

The *ARM® Toolkit Proprietary ELF* (ATPE) format is supported only by the RVDK for Philips Nexperia Sy.Sol compilation tools and debugger. The objects and images produced by the compilation tools cannot be used by other toolchains. ATPE images can only be loaded to a debug target using the RealView Debugger that is provided with RVDK for Philips Nexperia Sy.Sol.

2.2 RealView Compilation Tools v2.2

This section describes the *RealView Compilation Tools v2.2* (RVCT). It comprises build tools and utilities, together with supporting documentation and examples.

You can use RVCT to build C, C++, or ARM assembly language programs into applications that run on your ARM architecture-based RISC processors.

Compilation tools feature restrictions apply to both the command line of the tool and to #pragma entities.

The compilation tools are restricted to generate little endian code for the ARM926EJ-S™ and ARM946E-S™ processors only. The only supported floating-point models are `--fpu softvfp` (default) and `--fpu none`.

2.2.1 Components of RVCT

The RVCT consists of the following major components:

- *Compilation tools*
- *Utilities* on page 2-4
- *Supporting software* on page 2-5.

Compilation tools

The following compilation tools are provided:

armcc The ARM and Thumb® C and C++ compiler. The compiler is tested against the Plum Hall C Validation Suite for ISO conformance. It compiles:

- ISO C source into 32-bit ARM code
- ISO C++ source into 32-bit ARM code
- ISO C source into 16-bit Thumb code
- ISO C++ source into 16-bit Thumb code.

`armcc` creates only ATPE format objects.

armasm The ARM and Thumb assembler. This assembles both ARM assembly language and Thumb assembly language source. `armasm` creates only ATPE format objects.

armlink The ARM linker. This combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program. The ARM linker creates ATPE executable images.

Rogue Wave C++ library

The Rogue Wave library provides an implementation of the standard C++ library as defined in the *ISO/IEC 14822:1998 International Standard for C++*. For more information on the Rogue Wave library, see the online HTML documentation on the CDROM.

support libraries

The ARM C libraries provide additional components to enable support for C++ and to compile code for different architectures and processors.

Utilities

The following utility tools are provided to support the main development tools:

fromELF The ARM image conversion utility. This accepts ATPE input files and converts them to a variety of output formats, including:

- plain binary
- Motorola 32-bit S-record format
- Intel Hex 32 format
- Verilog-like hex format.

`fromELF` can translate ATPE images into formats suitable for FLASH/ROM programmers. Apart from this, it cannot translate ATPE objects or images into formats suitable for use by other versions of the ARM toolchains.

`fromELF` can also generate text information about the input image, such as code and data size.

This toolkit version of `fromELF` does not support certain features, compared with the full `fromELF`. See the chapter on using fromELF in *RealView Developer Kit v2.2 Linker and Utilities Guide* for more information.

armar The ARM librarian enables sets of ATPE object files to be collected together and maintained in libraries. You can pass such a library to the linker in place of several ATPE files.

Supported standards

The industry standards supported by RVCT include:

ar UNIX-style archive files are supported by `armar`.

DWARF2 DWARF2 debug tables are supported by the compiler, linker, and RealView Debugger.

ISO C The ARM compiler accepts ISO C as input. The option `--strict` can be used to enforce strict ISO compliance.

ISO C++ The ARM compiler supports the full ISO C++ language.

ABI for the ARM Architecture

The *Application Binary Interface for the ARM Architecture* (ABI for the ARM Architecture). For more details, see `http://www.arm.com`. RVDK generates ATPE objects, which are not ABI-compliant. RVDK can consume both ATPE objects and ABI-compliant ELF objects.

Supporting software

To debug your programs with RealView ICE Micro Edition connected to your evaluation board, use RealView Debugger.

2.3 RealView Debugger v1.8

This section describes the features available in RealView Debugger v1.8. It contains the following sections:

- *RealView Debugger concepts and terminology*
- *OS awareness* on page 2-7
- *Extended Target Visibility (ETV)* on page 2-7
- *Project Manager* on page 2-7
- *RealView Debugger downloads* on page 2-8
- *RTOS support* on page 2-8.

——— **Note** ———

RealView Debugger can load ATPE format images generated by `armlink`, flash/ROM images generated by `fromELF`, or third-party ELF images. The debugger will only work when connected to a permitted processor, using the RVI/RVI-ME provided with RVDK. Refer to *RealView Developer Kit licensing* on page 1-6 for further information.

No RVI unit is provided with the toolkit.

———————————

2.3.1 RealView Debugger concepts and terminology

The following terminology is used throughout the RVDK documentation suite to describe debugging concepts:

Debug target

A piece of hardware or simulator that runs your application program. A hardware debug target might be a single processor, or a development board containing a number of processors. However, if you have a multiprocessor board, you can only connect to one processor at a time.

Connection The link between RealView Debugger and the debug target.

Project A project is the highest level structural element that you can use to organize your source files and determine their output. You can use RealView Debugger to:

- create a range of software projects using predefined templates included in the root installation
- access image-related settings through auto-projects
- view and change project properties
- define different build target configurations

- set up a project environment automatically when the workspace opens
- open projects automatically when you connect to a specified debug target.

RTOS Operating systems provide software support for application programs running on a target. *Real Time Operating Systems* (RTOSs) are operating systems that are designed for systems that interact with real-world activities where time is critical.

Multithreaded operation

RTOS processes can share the memory of the processor so that each can share all the data and code of the others. These are called *threads*. RealView Debugger enables you to:

- attach Code windows to threads to monitor one or more threads
- select individual threads to display the registers, variables, and code related to that thread
- change the register and variable values for individual threads.

2.3.2 OS awareness

RealView Debugger v1.8 enables you to:

- use RTOS debug including *Halted System Debug* (HSD)
- interrogate and display resources after execution has halted
- access semaphores and queues
- view the status of the current thread or other threads
- customize views of application threads.

2.3.3 Extended Target Visibility (ETV)

RealView Debugger v1.8 provides visibility of targets such as boards and SoC. You can configure targets using board-chip definition files and preconfigured files are available:

- ARM family files provided as part of the installation
- customer/partner board files provided through ARM web resources at `http://www.arm.com`.

2.3.4 Project Manager

RealView Debugger v1.8 is a fully-featured *Integrated Development Environment* (IDE) including a project manager and build system.

2.3.5 RealView Debugger downloads

ARM provides a range of services to support developers using RealView Debugger. Among the downloads available are enhanced support for different hardware platforms through technical information and board description files. See `http://www.arm.com` to access these resources.

2.3.6 BCD Files

This toolkit is not supplied with any BCD or FME files, although these may be available separately from your supplier.

2.3.7 RTOS support

You must obtain the RealView Debugger support package for the RTOS you are using before you can use this extension. Select **Goto RealView RTOS Awareness Downloads** from the Code window **Help** menu for information on how to do this.

The RTOS support chapter in the *RealView Developer Kit v2.2 Debugger User Guide* shows how to use the thread drop-down list in the Code window and the additional tabs available in the Resource Viewer window. Using these facilities, you can:

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

Note

It is recommended that you read the section on Using RealView Debugger RTOS support in the *RealView Developer Kit v2.2 Debugger User Guide* before attempting to debug an RTOS using RealView Debugger. This section provides two examples of debugging an RTOS, and assumes you have experience with using RealView Debugger only for single-threaded programs.

2.4 RealView ICE Micro Edition v1.1

RealView ICE Micro Edition is provided to enable you to debug software running on the development boards supported by RVDK. It communicates through the EmbeddedICE® logic contained in the related processors.

See the *RealView ICE Micro Edition v1.1 User Guide* for more details on using and configuring RVI-ME.

2.5 RealView ICE

RealView ICE is an EmbeddedICE® logic debug solution. It enables you to debug software running on ARM processor cores that include the EmbeddedICE logic, and on appropriate *Digital Signal Processor* (DSP) cores.

RealView ICE provides the software and hardware interface between a debugger running on a Windows, Red Hat Linux or Sun Solaris host computer, and a *Joint Test Action Group* (JTAG) *IEEE Standard 1149.1* port on the target hardware.

The minimum RealView ICE version that should be used with your debugger is Version 1.5. RealView ICE is not supplied with the toolkit, and if you wish to make use of trace functionality you must additionally purchase the RealView ICE and RealView Trace products.

See the *RealView ICE and RealView Trace v1.5 User Guide* for more details on using and configuring RVI.

Chapter 3
RealView Debugger Desktop

This chapter describes, in detail, the RealView® Debugger desktop. It describes the contents of the default Code window, and explains how to change them. This chapter describes items and options available from the main menu and the toolbars.

It contains the following sections:

3.1 Basic elements of the desktop

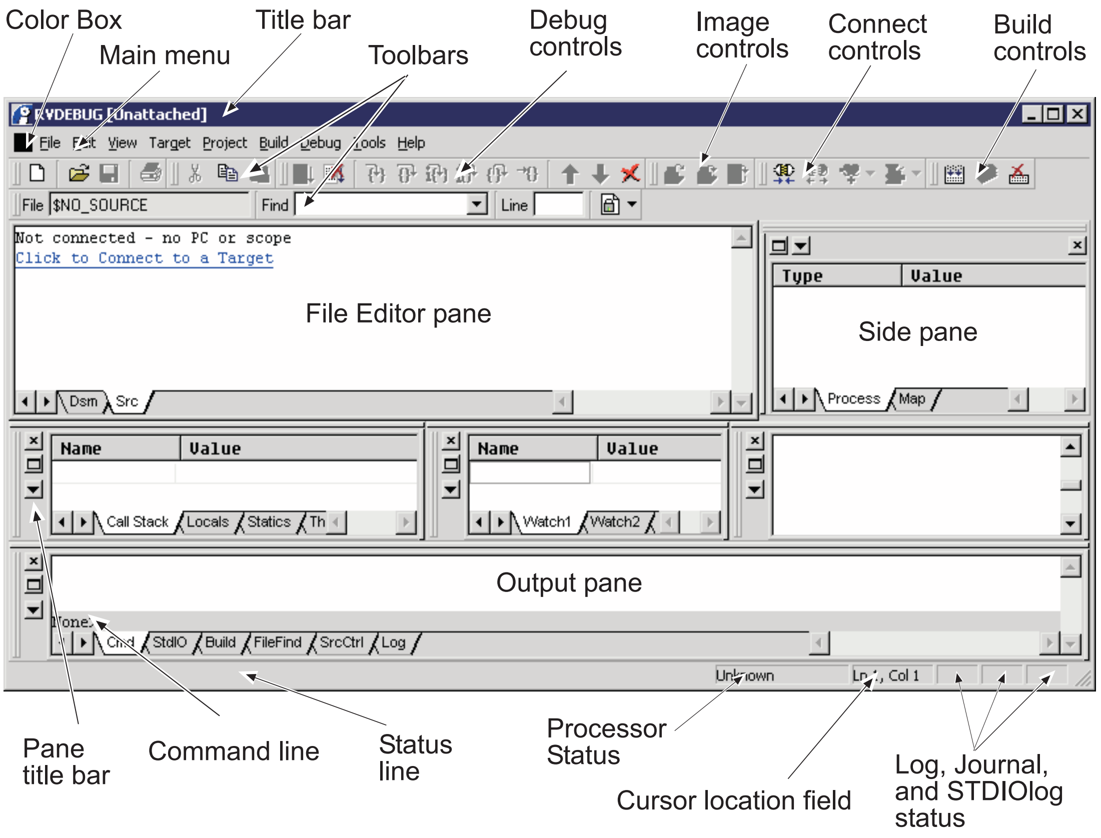
This section describes the default Windows desktop that you see when you run RealView Debugger for the first time after installation and highlights any key features that might be different. It contains the following sections:

- *Code window* on page 3-3
- *Default pane configuration* on page 3-5
- *Debug views and panes* on page 3-7
- *Pane controls* on page 3-9
- *Button toolbars* on page 3-11
- *Color Box* on page 3-12
- *Other window elements* on page 3-12.

3.1.1 Code window

When you run RealView Debugger for the first time after installation, the RealView Debugger Code window appears as shown in Figure 3-1.

When you exit RealView Debugger, it remembers the configuration of the panes or views by saving the current state in your workspace.



Not connected - no PC or scope
Click to Connect to a Target

None

Figure 3-1 Default Code window

The Code window is your main debugging and editing window. The contents of this window change as you:

- connect to targets
- load and unload application programs or files
- configure and customize your working environment.

The appearance of the Code window also depends on your licenses. For example, the base product enables you to debug your images in single connection mode, that is, where there is only one connection.

Title bar

The Code window title bar gives details of the current project, the current connection, and any processes running on your debug target. In addition to the application icon, the title bar contains (from left to right):

`RVDEBUG` Identifies the Code window. This changes as you open new windows, for example `RVDEBUG_1`, or `RVDEBUG_2`.

(*project*) The project associated with the image that you loaded.

In RealView Debugger, a project can be associated with a connection, that is it is *bound* to that connection. This is indicated by enclosing the project name in parentheses, for example, `(dhrystone)`.

Where a project is not associated with a particular connection, it is *unbound*. In this case, the project name is enclosed in angled brackets, for example `<my_project>`.

See the chapter that describes project binding in *RealView Developer Kit v2.2 Project Management User Guide* for more details.

Also see *Automatic operations performed by a project* on page 4-25.

@*endpoint_connection*:*execution_vehicle*

The connection, including the endpoint connection (usually the target processor) and the execution vehicle. For example, if you connect to an ARM946E-S processor through RealView ICE the title bar shows:

`@ARM946E-S_0:ARM-ARM-USB`

If you are using an RTOS, and you stop execution, details of the current thread replace the connection details. For example,

`T0x19F84_ITCM.ARM`

See the chapter that describes RTOS support in the *RealView Developer Kit v2.2 Debugger User Guide* for details.

`[Unattached]`

The attachment of the window to a specific connection or thread.

In single-processor debugging mode, this part of the title bar is blank and the option to attach windows to your connection is not available.

If you are using an RTOS, you can also attach a Code window to a thread. Only the debug information for the attached thread is displayed in that Code window. If a thread is attached to a Code window, the `[Unattached]` text is removed from the title bar.

See the chapter that describes RTOS support in *RealView Developer Kit v2.2 Debugger User Guide* for details.

3.1.2 Default pane configuration

The default pane configuration in the Code window, shown in Figure 3-1 on page 3-3, contains:

Top pane row

The top pane row contains the File Editor pane and the Process Control pane as a side pane by default.

The File Editor pane is always positioned in the top pane row. However, you can dock one or more other panes to the left or right of the File Editor pane.

File Editor pane

The File Editor pane is always visible when working with RealView Debugger. You cannot replace the File Editor pane with another pane (see *Debug views and panes* on page 3-7). However, you can define how the view is formatted, for example you can change the size of text displayed in the File Editor pane.

Use this area of the Code window to:

- use a shortcut to connect to a target or load an image.
- enter text to create project files
- open source files for editing and resaving
- view disassembly
- set breakpoints to control execution
- use the available menu options to search for specific text as part of debugging
- follow execution through a sequence of source-level and disassembly-level views

The File Editor pane contains a hyperlink to make your first connection to a debug target. When a connection is made, this link changes to give you a quick way to load an image.

When RealView Debugger first starts, the File Editor pane contains tabs to track program execution:

- the **Src** tab shows the current context in the source view
- the **Dsm** tab displays disassembled code with intermixed C/C++ source lines.

If you load an image, or when you are working with source files, more tabs are displayed, for example `dhry_1.c`. In this case, click on the **Src** tab to see the location of the PC.

See the chapter that describes editing source code in *RealView Developer Kit v2.2 Project Management User Guide* for full details on using the editing facilities in RealView Debugger.

Side pane By default, this contains the Process Control pane that displays details of the current process when you are connected to target processor. When you first run RealView Debugger, this pane is positioned to the right of the File Editor pane but you can float this pane, and place it at another position on the desktop.

The Process Control pane displays a **Threads** tab if you are debugging an RTOS application.

Middle pane row

The middle pane row contains the following panes by default:

Call Stack pane

Use this pane to:

- display the procedure calling chain from the entry point to the current procedure
- monitor local variables.

The Call Stack pane contains tabs:

Call Stack

Displays the stack functions call chain. This tab is selected by default.

Locals Shows variables local to the current function.

Statics Displays a list of static variables local to the current module.

This Shows objects located by the C++ specific `this` pointer.

Watch pane

Use this pane to:

- set up variables or expressions to watch
- display current watches
- modify watches already set
- delete existing watches.

The Watch pane contains tabs to display sets of watched values. The first tab, **Watch1**, is selected by default.

Memory pane

Use this pane to:

- display the contents of a range of memory locations on the target
- edit the values stored by the application.

Bottom pane row

The bottom pane row of the default Code window contains the Output pane by default. Select the different tabs to:

- enter commands during a debugging session (**Cmd**)
- handle I/O with your application (**StdIO**)
- see the progress of builds (**Build)**
- see the results of Find in Files operations (**FileFind**)
- see the results of operations using your version control tool (**SrcCtrl**)
- view the results of commands and track events during debugging (**Log**).

The command line is located at the bottom of the Output pane. The command prompt includes the status of the current process, for example, `Stop>`, `Run>`, or `None>` (no process). You can also enter debugger commands at this prompt.

3.1.3 Debug views and panes

RealView Debugger provides a range of debug views accessible through panes:

- Break/Tracepoints
- Call Stack
- Data Navigator
- Memory
- Process Control

- Registers
- Stack
- Symbol Browser
- Watch
- Locals tab of Call Stack pane
- Map tab of Process Control pane
- Threads tab of Process Control pane (with an RTOS extension)
- Output pane.

Substituting panes

All panes except for the File Editor pane can be substituted for another pane. To substitute one pane for another, select the required pane from the **Pane Control** menu of the pane to be substituted (see *Pane controls* on page 3-9 for details).

See the chapter that describes working with debug views in *RealView Developer Kit v2.2 Debugger User Guide* for more details on working with panes.

Adding panes

Select **View** to display the **View** menu to add another pane.

Initially, when you select a pane from the **View** menu it is displayed as a floating pane. If you select a pane that is already visible, then another instance of that pane is displayed, and the name has the suffix `_n`, where `n` is a number starting at one (for example, `Memory_1`).

If you choose to dock a pane, then a Select Location dialog box is displayed where you can specify the position to place the pane on the desktop.

During your debugging session, you can define which view is displayed in a chosen pane (see *Substituting panes* on page 3-8 for details).

See *Floating, docking, and resizing windows and panes* on page 3-21 for more details on floating and docking panes.

3.1.4 Pane controls

Each configurable pane in the Code window, shown in Figure 3-1 on page 3-3, includes a title bar and pane controls. In the side pane, the pane title bar is displayed horizontally at the top of the pane. In the middle and bottom panes, the title bar is displayed vertically at the left side of the pane.

When you float a pane, the title bar is displayed horizontally at the top of the window, and the title of the window is shown. Docked panes do not show the pane title, but see the description of the gripper bar in this section.

A pane contains the controls:

Pane Control

Click this button to display the **Pane Control** menu where you can change the debug view in the pane and rename the pane.

The top of the menu shows a list of any panes that you have previously viewed and hidden. You can select one of these panes, or select a pane from the **New Pane** submenu.

Visual controls

The visual controls are at the bottom of the **Pane Control** menu. Use these to:

- float the pane if it is docked
- dock the pane if it is floating
- hide the pane (see *Hidden panes* on page 3-10).

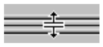
See *Floating, docking, and resizing windows and panes* on page 3-21 for more details.

Pane menu Click this button to display the **Pane** menu.

Use this to:

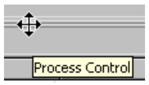
- change the display format
- change how pane contents are updated
- extract data from the pane.

The options available from this menu depend on the pane.



Resize control

Move the mouse to the edge of the pane to resize. When the resize cursor is displayed, click and drag the edge to resize.



Gripper bar

Move the mouse to the edge of the pane containing the gripper bar. In the side pane, the gripper bar is displayed horizontally at the top of the pane. In the middle and bottom panes, the gripper bar is displayed vertically at the left side of the pane.

When the gripper bar cursor is displayed, click and drag the pane to the new position. To float the pane, move the pane outside the RealView Debugger desktop.

If you hover the cursor over the gripper bar, the pane title is displayed as a tooltip. This is not affected by the state of the **Edit** → **Advanced** → **Tooltip Evaluation** option.

Hidden panes

A pane is classed as hidden if you change an existing pane to another type of pane, or you choose to hide it with the **Hide** option on this menu. The hidden state of a pane persists between RealView Debugger sessions.

To remove the hidden status of a pane:

1. Right-click on the toolbar of a pane that is currently displayed.

   The context menu shows visible panes as checked, and hidden panes as unchecked.

2. Select the required hidden pane:

   - if the current pane is docked, then the hidden pane is displayed docked, and to the left of the current pane.

    - if the current pane is floating, then the hidden pane is displayed floating, and overlays the current pane.

3. Click the **x** button on the pane toolbar to close the pane. That pane is no longer hidden, and so does not appear on any of the pane menus.

Hiding and restoring the Output pane

Only one instance of the Output pane can exist for a Code window. You can hide the Output pane by selecting **Hide** from the **Pane Control** menu. Also, if you click on the **x** button of the Output pane, the Output pane is hidden and not closed.

To make a hidden Output pane visible, either:

- perform steps 1 and 2 of the procedure described in *Hidden panes* on page 3-10
- select **View** → **Output** from the Code window main menu.

  Note

  The focus is on the tab that was visible at the time Output pane was closed. If the Output pane is open, this option has no effect.

If the Output pane is hidden during program execution, then it is made visible if your program requests user input. The pane is displayed in the same state (floating or docked) as when it was hidden.

3.1.5 Button toolbars

Below the Code window main menu, there are groups of toolbars that provide quick access to many of the features available from menu options, shown in Figure 3-2.

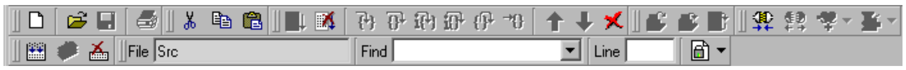


Figure 3-2 Code window toolbar

To hide a toolbar, right-click on any toolbar button. This displays the **Toolbars** menu where you can specify which toolbars are visible.

You can move a toolbar from the default position in the Code window so that it floats on your desktop, for example, the Debug toolbar shown in Figure 3-3. To restore the floating toolbar, double-click anywhere on the toolbar title bar.

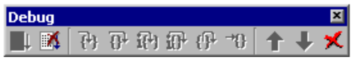


Figure 3-3 Debug toolbar (floating)

Repositioning a toolbar in this way applies only to the calling Code window. If you open a new Code window the toolbars are in the default positions.

See *Working with the Code window toolbars* on page 3-17 for details of the buttons available from the Code window toolbar, and how to customize toolbars.

3.1.6 Color Box

Code windows in RealView Debugger are color-coded to help with navigation. This is particularly useful when working with multithreaded applications.

The Color Box, shown in Figure 3-4, identifies the connection associated with the current window.

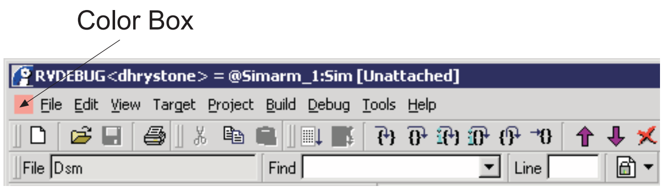

Color Box
RVDEBUG<dhrystone> = @Simarm_1:Sim [Unattached]
File Edit View Target Project Build Debug Tools Help
File Dsm
Find
Line

Figure 3-4 Color Box

When you first start RealView Debugger the Code window is not associated with any target or process. The Color Box changes when you make your first connection. As you create new Code windows, these are also color-coded.

Closing your connection changes the Color Box to show that there is no connection associated with the window. Any other Code windows attached to that connection are also updated to match.

When you are working with the Color Box, notice that:

- connection-independent windows, or controls, do not contain a Color Box
- floating panes contain a Color Box that matches the calling window.

3.1.7 Other window elements

There are status display areas at the bottom of the Code window, shown in Figure 3-1 on page 3-3:

Status line As you move through menu options, or when you view button tooltips, this line shows a more detailed explanation of the option or button under the cursor. When loading an image, it also shows the progress of the load.

Processor status

Indicates the processor status, which is one of `Unknown`, `Running`, or `Stopped`.

Cursor location field

As you move the cursor through a file within the File Editor pane, this shows the current location of the text insertion point.

LOG If this appears to the right of the Cursor location field, this shows that output is being written to a log file.

JOU If this appears to the right of the Cursor location field, this shows that output is being written to a journal file.

STDIOlog If this appears to the right of the Cursor location field, this shows that STDIO output is also being written to a STDIOlog file.

3.2 Finding options on the main menu

This section provides a summary of the main menu options available from the Code window (see Figure 3-1 on page 3-3) that enable you to:

- open and close files within the File Editor pane
- manage target images and connections
- manage projects
- build images
- manage workspaces
- navigate, search, and edit source files
- manage new windows and change pane contents
- debug your images
- access the online help system.

Note

Some menu options are not enabled unless you have suitable support, for example the **View** → **Threads tab** option.

The menus available from the main menu bar are:

File Displays the **File** menu.

The **File** menu also enables you to examine, and change, your workspace settings.

When you are using the File menu the menu option **Close Window** is not enabled when the default Code window is the only window. This is the main debugging and editing window, and it must be open throughout your debugging session. Close the Code window to close down RealView Debugger.

Edit Displays the **Edit** menu. This menu enables you to work with source files as you develop your application. It includes options to:

- define how source code is displayed in the File Editor pane.
- enable you to work with source files and perform searches on those files as you debug your image
- access the various browsers in RealView Debugger.

View Displays the **View** menu. This menu enables you to set up new windows and panes as you are working with target connections.

To use custom windows, you must have an appropriate third-party custom window plugin. A custom window plugin can be designed to start automatically when RealView Debugger starts. If you only have plugins of this type, then the **Custom Windows** option remains disabled. Otherwise, if a plugin is designed not to start automatically, then the **Custom Windows** option is enabled. In addition, an option is added to the submenu to enable you to start the custom window plugin. For instructions on installing third-party custom window plugins, and how to use them, see your Vendor documentation.

Target Displays the **Target** menu. This menu enables you to connect and disconnect from debug targets, configure your connections, load images, and attach the Code window to a connection.

Project Displays the **Project** menu. This menu enables you to work with projects so that you can organize your source files, model your build process, and share files with other developers. This can be used in conjunction with the **Build** menu to rebuild images.

These features are described in full in the chapter that describes working with projects in *RealView Developer Kit v2.2 Project Management User Guide*.

Build Displays the **Build** menu. This menu enables you to build files, or groups of files, to create your image ready for loading to a target.

For details of this menu see the chapter that describes building applications in *RealView Developer Kit v2.2 Project Management User Guide*.

Debug Displays the **Debug** menu. This menu includes the main facilities you use during a debugging session, such as setting up breakpoints and tracepoints, controlling program execution, and program Flash.

Tools Displays the **Tools** menu. This menu enables you to:

- set up Analyzer and Tracing features
- add and configure macros for use during debugging
- log information to the Log, Journal and STDIO log files
- run CLI commands from script files
- examine, and change, your global configuration options.

The **Tools** menu also provides access to options for Analysis and Profiling.

Note

Supported by selected simulators from Ceva Inc., the **Simulation Control** option is not available in this release.

Help Displays the **Help** menu.

This menu gives you access to the RealView Debugger online help, to web downloads pages, and displays details of your version of RealView Debugger. You can also use this menu to create and submit a *Software Problem Report* (SPR).

3.3 Working with the Code window toolbars

The Code window toolbars (see Figure 3-2 on page 3-11) give access to many of the features available from the main menu and to additional debugging controls. By default, the Code window shows all toolbars but you can customize which controls are available.

When working with Code window toolbars, use Windows **Tooltips** to see hover-style help when you hold your mouse pointer over a button.

This section includes:

- *File toolbar*
- *Edit toolbar*
- *Debug toolbar*
- *Image toolbar* on page 3-18
- *Connect toolbar* on page 3-18
- *Build toolbar* on page 3-19
- *Find toolbar* on page 3-19
- *Customizing the toolbars* on page 3-20.

3.3.1 File toolbar

Use these buttons when developing applications. They enable you to open files and to save changed files in the File Editor pane. These buttons replicate selected options from the **File** menu.

3.3.2 Edit toolbar

Use these buttons to edit source files in the File Editor pane. They replicate selected options from the **Edit** menu.

3.3.3 Debug toolbar

These buttons replicate selected options from the **Debug** menu. This toolbar consists of the following controls:

Execution controls

Use these buttons to start and stop program execution.

Stepping controls

Use these buttons to step though a program.

Context controls

Use these buttons to move up and down the stack levels during program execution. These buttons are enabled when an image has been loaded.

Command cancel

Commands submitted to RealView Debugger are queued for execution. Click this button to cancel the last command entered onto the queue.

This does not take effect until the previous command has completed.

3.3.4 Image toolbar

Use these buttons to control images. They replicate selected options from the **Target** menu.

3.3.5 Connect toolbar

Use these buttons to control target connections. They replicate selected options from the **Target** menu.

The **Cycle Connections** button allows you to switch to the next available active connection, but is only enabled during a multiprocessor debugging session. Click on the drop-down arrow to display a connection menu. This menu lists the active connections with the current connection marked with an asterisk. The menu also provides an option to attach the Code window to the current connection. When a Code window is attached to a connection a tick mark is added to the **Attach Window to Connection** option and the connection that is attached to the Code window. Although you can still cycle through multiple connections, the connection details do not change in Code windows that are attached.

The following button is also part of the Connect toolbar:

Cycle Threads button

Used during a multithreading debugging session, click this button to change to the next active thread. Click on the drop-down arrow to display the list of active threads where you can identify the current thread. The list also shows if the Code window is attached to this thread by adding a tick mark.

This button is only enabled when an underlying operating system is supported. See the chapter that describes RTOS support in *RealView Developer Kit v2.2 Debugger User Guide*.

3.3.6 Build toolbar

Use these buttons to control the build process during your debugging session. They replicate selected options from the **Build** menu.

3.3.7 Find toolbar

This toolbar becomes active when you are editing a file in the File Editor pane. It contains:

File This read-only field shows the name of the file currently displayed in the File Editor pane. If you have changed the file since loading or saving, an asterisk, * , is appended to the end of the filename.

If you are working on several files in the File Editor pane, the File field shows the name displayed on the topmost file tab.

Find This field enables you to perform a quick text search on the file currently displayed in the File Editor pane. Type the required string into the Find field and then press Enter. If you are working on several files in the File Editor pane, the search examines only the file in the topmost file tab. The search behavior is defined by the settings in the Find dialog box displayed from the **Find** menu.

Click on the drop-down arrow to display a list of recently used search strings.

Line Use the Line number field to enter the number of the line where the text insertion point is to be moved.

Source control button

This button indicates the read/write status of the current file. Click this button to change the status of a file.

You can edit a file only if the Read-Write icon is displayed (shown here).

Click on the drop-down arrow to access command menu:

- If your sources are not under version control, then this menu enables you to make the current source file in the File Editor pane read-only or editable.
- If your sources are under version control, this button shows the source control commands. See the chapter that describes working with version control systems in the *RealView Developer Kit v2.2 Project Management User Guide* for more details.

Encoding ASCII

Encoding Available only when you enable Internationalization. Click the down arrow and select the encoding format you require.

This enables you to view any multibyte strings in a source tab of the File Editor pane in the correct character encoding. It also enables you to search for multibyte text in your sources that matches the selected encoding.

3.3.8 Customizing the toolbars

You can customize the toolbars as described in the following sections:

- *Showing and hiding toolbars*
- *Floating and docking toolbars*.

Showing and hiding toolbars

To show or hide a toolbar:

1. Right-click on any toolbar button or the toolbar background to display the **Toolbars** menu.

   Checked options indicate which toolbars are currently shown on the RealView Debugger toolbar. All toolbars are shown by default.

2. Select the toolbar that you want to show or hide. A hidden toolbar is removed from the RealView Debugger toolbar.

Note

If you open a new Code window the toolbars are in the default positions.

Floating and docking toolbars

To float a toolbar:

1. Click on the toolbar control.

2. Drag and drop the toolbar on your desktop.

To dock a floating toolbar, double-click on the toolbar title.

3.4 Working in the Code window

This section describes how to work with the Code window:

- *Floating, docking, and resizing windows and panes*
- *Changing the focus* on page 3-24
- *In-place editing* on page 3-24
- *Working with tabs* on page 3-25
- *Working with scroll bars* on page 3-25.

3.4.1 Floating, docking, and resizing windows and panes

Panes are docked to default positions and have default sizes in the Code window when RealView Debugger starts in the default state.

Resizing docked panes

To change the height of a docked pane in the middle or bottom rows, drag the upper boundary of the row containing the pane to the required height. All panes in the row are resized. This also changes the height of the panes in the first row.

To change the width of a pane, drag the left boundary of the pane to the new position.

Floating a pane

A pane is floating when it is displayed separately from the calling window and can be moved around the desktop. To float a pane:

- select **Float** from the **Pane Control** menu
- click the gripper control, and drag the pane to the required position on your desktop
- double-click on the pane gripper control.

A floating pane is still tied to the calling window, as shown by the Color Box (see *Color Box* on page 3-12).

Docking a pane

To dock a floating pane:

- Select **Dock** from the **Pane Control** menu. If the pane has not been docked previously, a Select Location dialog box is displayed. You can choose to dock the pane to the left, right, or bottom of the Code window.

- click the pane title bar, and drag the pane to the required position on the Code window
- double-click on the pane title bar.

Pane docking behavior

When you dock a pane, the pane is positioned as described in the following sections.

Using the Select Location dialog box to dock a pane

For a pane that has not previously been docked, a Select Location dialog box is displayed containing the following options:

Left The pane is docked to the left of the File Editor pane. If a pane already exists to the left of the File Editor pane, then the pane is docked below that pane.

Right The pane is docked to the right of the File Editor pane. If a pane already exists to the right of the File Editor pane, then the pane is docked below that pane.

Bottom The pane is docked to create a new bottom pane row.

Dragging and dropping on a Code window border

When you drag and drop a pane on the Code window border:

Left border

The pane is docked as the left-most pane in the top row.

Right border

The pane is docked as the right-most pane in the top row.

Bottom border

The pane is docked to create a new bottom pane row.

Dragging and dropping on a middle or bottom row pane border

When you drag and drop a pane on the border of another pane that is not in the top row:

Left border

The pane is docked in the that row, and to the left of the currently docked pane.

Right border

The pane is docked in the that row, and to the right of the currently docked pane.

Bottom border

The pane is docked in the that row, and above the currently docked pane.

Top border

The pane is docked in the that row, and to the left or right of the currently docked pane, depending on which side of the pane center you drop it.

Dragging and dropping on a top row pane border

When you drag and drop a pane on the border of another pane in the top row:

Left border

The pane is docked in the top row, and to the left of the currently docked pane.

Right border

The pane is docked in the top row, and to the right of the currently docked pane.

Bottom border

The pane is docked in the top row, and below the currently docked pane. However, if you drop it on the bottom of the File Editor pane, then the pane is docked as a new row, between the top row and the row below it.

Top border

The pane is docked in the top row, and above the currently docked pane.

Docking a pane that was previously docked

If the floating pane was previously docked, then the docking behavior depends on whether or not the row that originally contained the pane still exists:

- If the row still exists, then the pane is restored to its original position in that row.
- If the row no longer exists, then the pane is restored to a position in the bottom row. Usually, this is to the left of that row, but this depends on whether or not you have undocked and docked other panes in that row.

3.4.2 Changing the focus

In RealView Debugger, the *focus* indicates the window, or pane, where the next keyboard input takes effect. Use a single left-click on the title bar to move the focus to the Code window, or the pane, where you want to work. If the focus is currently in a source file, you can move the focus from the source file to the Output pane by pressing Ctrl+Tab or Ctrl+Shift+Tab.

When working with several Code windows, left-click inside a pane entry to change the focus. If the pane is floating, the pane title bar changes color to show that it now has the focus, and the title bar of the calling Code window is also highlighted.

If you switch to another Code window by clicking on the title bar, the focus moves to that window and the text insertion point is located inside the File Editor pane. If the context of the Code window is unknown, the text insertion point is located at the command-line prompt.

If you double left-click in a pane entry, for example on the contents of a register in the Register pane, this moves the focus to this pane and highlights the entry ready for editing.

If you right-click in a pane that does not have the focus, the focus does not move to this pane. This action does, however, highlight the chosen entry in the new pane. In this case, use the Code window title bar to see where the focus is currently located.

3.4.3 In-place editing

In-place editing enables you to change a stored value and to see the results of that change instantly. RealView Debugger offers in-place editing whenever possible. For example, if you are displaying the contents of memory or registers and you want to change a stored value:

1. Double-click in the value you want to change, or press Enter if the item is already selected. The value is enclosed in a box with the characters highlighted to show they are selected (pending deletion).
2. Either:
    - enter data to overwrite the highlighted content
    - press the left or right arrow keys to deselect the existing data and position the insertion point where you want to make a change.
3. Press Enter to store the new value in the selected location.

If you press Escape before you press Enter, any changes you have made in the highlighted field are ignored.

In-place editing is not suitable for:

- editing complex data where some prompting is helpful
- editing groups of related items
- selecting values from predefined lists.

In these cases, an appropriate dialog box is displayed.

——— **Note** ———

When using in-place editing, you must either complete the entry and press Enter, or press Escape to cancel the operation. If you move the focus to another pane, RealView Debugger cancels the current in-place editing operation.

3.4.4 Working with tabs

You can access RealView Debugger debugging features using tabbed pages or *tabs*. In the Watch pane, for example, there are multiple tabs and each might show a different set of watched variables. The Output pane contains tabs enabling you to select the view that suits your debugging task.

Right-click on a tab to display text that explains the function of the tab or the content. If you are using the default Windows display settings and you right-click on a tab that is not at the front, the tab name being referenced is colored red for easy identification.

If you right-click over a blank area of the tab bar at the bottom of a window or pane, a context menu enables you to select a tab from a list. You can also display this menu by right-clicking on the left, or right, scroll arrow on the left-hand side of the tab bar.

3.4.5 Working with scroll bars

If you are working in a pane where vertical scroll bars are enabled, right-click on the up, or down, scroll arrow to see the **Scroll** menu. Using this menu you can scroll by a line or a page, or jump to the beginning or end of the related information.

Chapter 4
Getting Started with RealView Developer Kit

The component products provided with RVDK enable you to build and debug one or more images that make up your application. This chapter introduces you to the basic tasks for building and debugging with the RVDK tools. It contains the following sections:

4.1 Using the example projects

RVDK provides a number of example projects. See *RVDK example projects* on page 1-4.

The tasks described in the RealView Debugger and RVCT documentation use the Dhrystone project and some of the other example projects.

Until you are familiar with the features of RealView Debugger and RVCT, it is suggested that you follow the instructions as described.

The tasks described in this chapter use the Dhrystone example project where appropriate:

- The RVDK Dhrystone project is in the RVDK examples directory `...\dhrystone`.

  This contains a predefined RealView Debugger project file, and a precompiled image. This example project is used in the following RealView Debugger tasks:
  - *Loading an image ready for debugging* on page 4-10
  - *Basic debugging tasks with RealView Debugger* on page 4-14
  - *Building and rebuilding an image with RealView Debugger* on page 4-21.

4.2 RealView ICE Micro Edition Connection Sequence

To ensure proper RVDK operation, perform the following steps. Refer to your *RealView ICE Micro Edition v1.1 User Guide* for further information.

1. Ensure your processor board is powered up.

2. Connect the RVI-ME to the board using the JTAG interface cable.

3. Connect the RVI-ME to your PC using a USB cable. Windows may ask you to install a driver for RVI-ME. The driver, `RVI-ME.sys`, can be found in the root directory of the RVDK CD. Ensure the RVI-ME power indicator is illuminated. If it fails to illuminate, disconnect and reconnect the USB cable.

   If you have any problems connecting your RVI-ME, contact your supplier.

4.3 RealView ICE Connection Sequence

If you've purchased an optional RVI unit, to connect the RVI run control unit to your host computer and to the target hardware, perform the following steps. Refer to your *RealView ICE and RealView Trace v1.5 User Guide* for more detailed information.

1. Ensure the RealView ICE software is installed on the host computer. See the *RealView ICE Installation Guide* for information on how to do this.
2. Connect the host computer to the RealView ICE run control unit, using either the USB port or a TCP/IP network connection, as required.
3. Connect the RealView ICE run control unit to the target hardware, using the appropriate cable.
4. If you are using RealView Trace, you have to connect the RealView Trace unit to the RealView ICE unit and to the target board.
5. Power up the target hardware.
6. Connect the external power supply to the RealView ICE run control unit, and to the mains electricity.
7. Switch on the power supply. The power LED and the expansion bus power LED both switch on.
8. The RealView ICE run control unit firmware is based on an embedded Linux kernel, so the unit takes a short time to boot up and establish either a network or USB connection.
9. If your RealView ICE unit is connected to a network, you must now run the Config IP application to configure the network settings. This is described in the chapter on configuring RealView ICE networking in *RealView ICE and RealView Trace v1.5 User Guide*.

   Note

   You have only to perform the network configuration once.

   If the RealView ICE unit is powered up with only a USB connection, it uses an IP address of 127.0.0.0. However, if a network cable is also attached, the IP address associated with the USB connection is the IP address that you have assigned to the RealView ICE unit, or that it obtains from a DHCP server.

   If you have any problems connecting your RealView ICE, contact your supplier.

4.4 Starting and Exiting RealView Debugger

This section includes:

- *Starting RealView Debugger*
- *Exiting RealView Debugger*.

4.4.1 Starting RealView Debugger

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

Programs → ARM → RealView Developer Kit v2.2 for Philips Nexperia Sy.Sol → RealView Debugger v1.8

——— **Note** ———

If you are using the default Windows XP settings, select **All Programs**.

————————

The RealView Debugger splash screen is displayed, and is then replaced by the RealView Debugger Code window, as shown in Figure 3-1 on page 3-3.

4.4.2 Exiting RealView Debugger

To exit RealView Debugger, select the menu option **File** → **Exit**.

When you exit RealView Debugger, the current state of your Code window and connection is saved. Therefore, the next time you start RealView Debugger:

- the Code window appears in the same state as your previous debugging session
- RealView Debugger automatically attempts to reconnect to a debug target, if it was previously connected when you last exited RealView Debugger.

4.5 Opening an existing RealView Debugger project

To open an existing RealView Debugger project file:

1. Start RealView Debugger (see *Starting RealView Debugger* on page 4-5).
2. Select the menu option **Project** → **Open Project...**. The Select Project to Open dialog is displayed.
3. Locate the project file (`.prj`) in the project directory.
4. Click **Open**. The project is loaded into RealView Debugger.

You can also arrange for RealView Debugger to:

- open a project automatically when you connect to a debug target (see *Making a project and a connection interdependent* on page 4-25 for details)
- perform automatic operations when a project is opened (see *Automatic operations performed by a project* on page 4-25 for details).

For more details on opening projects, see the chapter on managing projects in the *RealView Developer Kit v2.2 Debugger User Guide*.

4.6 Changing your target board/chip definition

By default, your RVI-ME debug target is preconfigured. However, other board and processor definitions might be provided that are more suitable to your specific debug target. If you want to use another board or processor definition, you must configure the RVI-ME debug target. This section shows you how to do this, but for more detailed information about configuring your debug targets, see the *RealView Developer Kit v2.2 Debugger User Guide*.

4.6.1 How to change the board/chip definition

To configure the debug target:

1. Display the Connection Control window.

   Select the menu option **Target** → **Connect to Target...**. The Connection Control window is displayed.

2. If RealView Debugger is connected to the debug target, click the check box for the target processor so that it is unchecked. This disconnects the debug target.

3. Select the menu option **Target** → **Connection Properties...**. The Connection Properties window is displayed.

4. In the left-hand pane, select the `CONNECTION=`*`connection_name`* entry, where *`connection_name`* is the name of the connection you want to modify.

5. In the right-hand pane, right-click on the blue `*BoardChip_name` setting that identifies the current board or processor name that you want to change.

6. Select the new board or processor name from the context menu. The value of the blue `*BoardChip_name` setting is updated.

7. Select the menu option **File** → **Save and Close** to save the changes.

8. Connect to the debug target.

4.7 Connecting to your target

Before you can load an image to your target, you must connect to it. This section describes the connection-related tasks. It contains:

- *Configurating your target connection*
- *Connecting to your target*
- *Setting connect mode* on page 4-9.

4.7.1 Configurating your target connection

This RVDK is provided with preconfigured target connections for use with RVI-ME. If you are using an optional RVI unit, you must configure the connection manually. However, you can configure the RVI-ME target connection to your specific requirements. See the chapters on configuring targets and connections in the *RealView Developer Kit v2.2 Debugger User Guide*.

This toolkit is not supplied with any BCD or FME files, although these may be available separately from your supplier. To configure board/chip files, see *How to change the board/chip definition* on page 4-7.

4.7.2 Connecting to your target

To connect to your target, do the following:

1. Select the main menu option **Target → Connect to Target...**.

   Alternatively, click on the blue `Click to Connect to a Target` hyperlink in the File Editor pane (see Figure 3-1 on page 3-3).

   The Connection Control window is displayed. If there are currently no connections to your target, the target execution vehicle (`ARM-ARM-NW` for RVI, or `ARM-ARM-USB` for RVI-ME) is expanded to show the available access-provider connections.

   For more details on connecting to a target, see the chapter on connecting to targets in the *RealView Developer Kit v2.2 Debugger User Guide*.

2. Expand the access-provider connection in the connection tree control to show the target processor available for the connection.

3. Click the check box for the target processor. The connection to your target is established, and the Code window is updated with details for the connection.

4.7.3 Setting connect mode

You can also control the way a target processor starts when you connect by setting the connect mode. See the chapter on connecting to targets in the *RealView Developer Kit v2.2 Debugger User Guide*.

4.8 Loading an image ready for debugging

This section describes how to get an image loaded on to a debug target. The instructions assume that you are familiar with the RealView Debugger desktop described in Chapter 3 *RealView Debugger Desktop*. It contains the following sections:

- *Loading an image directly*
- *Loading an image associated with a RealView Debugger project*
- *Loading multi-image applications* on page 4-11
- *Working with memory* on page 4-11
- *Reloading an image* on page 4-11.

4.8.1 Loading an image directly

To load an image directly to your debug target:

1. Start RealView Debugger (see *Starting RealView Debugger* on page 4-5).
2. Connect to the debug target (see *Connecting to your target* on page 4-8).
3. Select the menu option **Target → Load Image...**.

   Alternatively, click the blue `Click to Load Image to Target` hyperlink in the File Editor pane.

   The Load File to Target dialog is displayed.
4. Locate the `dhrystone.axf` image in the RealView Debugger examples directory:

   `examples_directory``\dhrystone\Debug`
5. Click **Open**. The image is loaded to the debug target.

   The image name is inserted as a project name in the Title bar of the Code window. RealView Debugger has created an auto-project. See the chapter on managing projects in the *RealView Developer Kit v2.2 Debugger User Guide* for more details on auto-projects.

   Also, the source file `dhry_1.c` is opened in the File Editor pane.

4.8.2 Loading an image associated with a RealView Debugger project

To load an image associated with a RealView Debugger project:

1. Start RealView Debugger (see *Starting RealView Debugger* on page 4-5).
2. Connect to the debug target (see *Connecting to your target* on page 4-8).
3. Open the RealView Debugger Dhrystone example project file (`dhrystone.prj`) located in the RealView Debugger examples directory:

`examples_directory\dhrystone`

See *Opening an existing RealView Debugger project* on page 4-6 for instructions on how to open a project.

4. To load the image to your connected debug target, click the blue `Click to Load image_path` hyperlink in the File Editor pane.

For more details on loading images, see the chapter on working with images in the *RealView Developer Kit v2.2 Debugger User Guide*.

4.8.3 Loading multi-image applications

If your application contains multiple images, for example an executable image and an RTOS image, you can load all the images to the target. You can load the first image either directly or from a RealView Debugger project (see *Loading an image associated with a RealView Debugger project* on page 4-10). However, you must load any subsequent images directly (see *Loading an image directly* on page 4-10). For each image that you load directly, you must ensure that the **Replace Existing File(s)** check box is not selected.

For more details about working with multiple images, see the chapter on working with images in the *RealView Developer Kit v2.2 Debugger User Guide*.

4.8.4 Working with memory

Before you load an image, you might have to define memory settings. This depends on the debug target you are using to run your image.

Where appropriate, defining memory gives you full access to all the memory on your debug target. RealView Debugger enables you to do this in different ways, for example using an include file, or defining the memory map as part of your target configuration settings.

For instructions on working with memory and memory maps, see the chapters describing memory mapping and reading and writing memory, registers, and flash in *RealView Developer Kit v2.2 Debugger User Guide*.

4.8.5 Reloading an image

During your debugging session you might have to amend your source code and then recompile. Select **Target → Reload Image to Target** from the Code window to reload an image following these changes.

Reloading an image refreshes any window displays and updates debugger resources.

4.9 Unloading an image

RealView Debugger automatically unloads an image from a debug target when you:

- disconnect from the debug target
- exit RealView Debugger.

Also, if you close the RealView Debugger project associated with an image, RealView Debugger displays a prompt asking if you want the image unloaded.

However, you might want to unload an image explicitly as part of your debugging session, for example if you correct coding errors and then rebuild outside RealView Debugger. See *How to explicitly unload an image* for instructions.

You do not have to unload an image from a debug target before loading a new image for execution. Display the Load File to Target dialog box and ensure that the **Replace Existing File(s)** check box is selected ready to load the next image (see *Loading an image directly* on page 4-10).

4.9.1 How to explicitly unload an image

You can unload an image by using the Process Control pane. To do this:

1. Select **View** → **Process Control** from the default Code window main menu.

   If you have still have the default panes visible in the Code window, as shown in Figure 3-1 on page 3-3, then the Registers pane is replaced by the Process Control pane.

2. Right-click on the `Image` entry, for example `dhrystone.axf`, or on the `Load` entry, `Image+Symbols`, to display the **Image** context menu.

3. Select **Unload**.

Alternatively, in the Process Control pane click on the check box associated with the `Load` entry so that it is not selected.

——— **Note** ———

Unloading an image does not affect target memory. It unloads the symbols and removes most of the image details from RealView Debugger. However, the image name is retained.

To remove image details completely, right-click on the `Image` entry in the Process Control pane and select **Delete Entry**.

———

4.10 Running the image

To run an image:

1. Load the image to your debug target (see *Loading an image ready for debugging* on page 4-10).
2. Either:
   - Select **Debug** → **Run** from the main menu.
   - Click the **Go** button on the Actions toolbar.

4.11 Basic debugging tasks with RealView Debugger

The basic debugging tasks you can perform with RealView Debugger are shown in Table 4-1. The references give detailed instructions for these tasks.

Table 4-1 Basic Debugging Tasks

Task	For detailed instructions, see
Displaying line numbers in the current source view	*Displaying line numbers* on page 4-15
Using breakpoints	*Setting a simple breakpoint* on page 4-15
Displaying variables	*Displaying variables* on page 4-19, and the chapter on working with browsers in the *RealView Developer Kit v2.2 Debugger User Guide*
Examining different code views	the chapter on controlling execution in the *RealView Developer Kit v2.2 Debugger User Guide*
Displaying register contents	the chapter on monitoring execution in the *RealView Developer Kit v2.2 Debugger User Guide*
Changing register contents	the chapter on reading and writing memory, registers, and flash in the *RealView Developer Kit v2.2 Debugger User Guide*
Using the Process Control pane	the chapter on working with images in the *RealView Developer Kit v2.2 Debugger User Guide*
Displaying memory contents	the chapter on monitoring execution in the *RealView Developer Kit v2.2 Debugger User Guide*
Using Tooltip Evaluation to examine variables and register contents	the chapter on editing source code in the *RealView Developer Kit v2.2 Debugger User Guide*
Using the call stack	the chapter on monitoring execution in the *RealView Developer Kit v2.2 Debugger User Guide*
Using browsers and lists	the chapter on working with browsers in the *RealView Developer Kit v2.2 Debugger User Guide*
Setting watches	the chapter on monitoring execution in the *RealView Developer Kit v2.2 Debugger User Guide*

4.11.1 Displaying line numbers

To display line numbers for the any source file displayed in the File Editor pane, select the menu option:

Edit → Editing Controls → Show Line Numbers

4.11.2 Setting a simple breakpoint

To set a simple, unconditional breakpoint:

1. Make sure line numbers are displayed (see *Displaying line numbers*).

   This is not necessary but might help you to follow the examples.

2. Right-click in the first entry in the Memory pane, `<NoAddr>`, and select **Set New Start Address...** from the context menu.

3. Enter a value as the start address for the area of interest, for example `0x8008`.

4. Click **Set** to confirm the setting and close the dialog box.

5. Click on the **Src** tab in the File Editor pane.

6. Set a simple, unconditional breakpoint at line 149 in `dhry_1.c`, `Proc_5();`. To do this double-click on the line number.

   If the line number is not visible, then double-click inside the gray area at the left of the required statement in the File Editor pane to set the breakpoint.

7. Set a watch on the variable `Int_1_Loc` at line 152 in `dhry_1.c`. To do this right-click on the variable. The variable is then underlined in red.

8. Select **Watch** from the context menu.

9. Start execution (see *Running the image* on page 4-13)

10. Enter the required number of runs, for example `50000`.

11. Monitor execution until the breakpoint is reached.

12. Click **Go** again and monitor the program as execution continues.

See the chapter on working with breakpoints in the *RealView Developer Kit v2.2 Debugger User Guide* for more details.

4.11.3 Setting the top of memory

Select **View** → **Registers** to open the Register pane at the default location. The Debugger Internals are available from the **Debug** tab inside this pane. To see a list of the available tabs, right click on the double arrows to the left of the Register pane tabs.

In RVDK, `top_of_memory` is used to set the application stack base for a semihosted application running on a remote target.

If `top_of_memory` is not set, RVDK sets it to a default value of `0x80000`. A warning is displayed in the **Cmd** and **Log** tabs:

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

The value of `top_of_memory` can be overridden for a single debug session from the **Debug** tab in the Register pane.

The value of `top_of_memory` can also be set for a particular target connection using the Connection Properties window:

1. Ensure that you disconnect from the target before making these changes.
2. Select **Target** → **Connect to Target...** to open the Connection Control window.
3. Right-click on the connection, for example `RVI-ME`, and select **Connection Properties...** from the context menu.

   The appropriate branch in the Connection Properties window opens automatically.
4. Drill down through the tree:
   - `Advanced_Information`
   - `Default`
   - `ARM_config`
5. Right-click on the `Top memory` entry and set the new connection default.
6. Select **File** → **Save and Close**.

——— **Note** ———

To use the new setting, you must now connect to the target.

If a Board/Chip definition file is selected for this connection, then this file might contain a value for `top_of_memory` that overrides the target connection setting.

———————————

4.11.4 Configuring Vector Catch

Vector Catch is a mechanism used to trap processor exceptions. This feature is typically used in the early stages of development to trap processor exceptions before the appropriate handlers are installed. You select the vectors to trap by editing the `vector_catch` value.

The default values correspond to trapping the exceptions listed in Table 4-2.

Table 4-2 Trapped Processor Exceptions (defaults)

Exception	Trapped	Comment
Reset	Yes	Set to True to catch Reset vectors. This is the default.
Undefined	Yes	Set to True to catch Undefined/Illegal Instructions. This is the default.
SWI	No	Set to True to catch software interrupts. The default is False. The SWI vector may also be trapped by the debugger to enable standard semihosting.
Prefetch Abort	Yes	Set to True to catch Prefetch abort (instruction fetch memory fault) exceptions. This is the default.
Data Abort	Yes	Set to `True` to catch Data abort (data access memory fault) exceptions. This is the default.
IRQ	No	Set to True to catch interrupt requests. The default is False.
FIQ	No	Set to True to catch fast interrupt requests. The default is False.

Setting Vector Catch for a Particular Connection

The state of individual vectors for catching can also be set for a particular target connection using the Connection Properties window:

1. Ensure that you disconnect from the target before making these changes.
2. Select **Target** → **Connect to Target...** to open the Connection Control window.
3. Right-click on the connection, for example `RVI-ME`, and select **Connection Properties...** from the context menu.

   The appropriate branch in the Connection Properties window opens automatically.

4. Drill down through the tree:
   - `Advanced_Information`
   - `Default`
   - `ARM_config`
   - `Vectors`
5. Set the catching of each vector in the group to the required new connection default.
6. Select **File** → **Save and Close**.

——— **Note** ———

To use the new setting, you must now connect to the target.

———————————

Setting Vector Catch for Individual Target Boards

You can set the default vector catch that will be used for a particular target board, using the Connection Properties window:

1. Ensure that you disconnect from the target before making these changes.
2. Select **Target** → **Connect to Target** to open the Connection Properties window.
3. Select the appropriate board from within the Board/Chip Definitions folder
4. Select the appropriate board from within the Board/Chip Definitions folder and drill down through the tree:
   - `Advanced_Information`
   - `ARM`
   - `ARM_config`
   - `Vectors`
5. Set the catching of each vector in the group to the required board default.
6. Select **File** → **Save and Close**.

——— **Note** ———

Vector settings in the Board/Chip definition file will override those in the target connection settings.

———————————

Setting Vector Catch from the Debug Menu

You can override the value of `vector_catch` for a single debug session as follows:

1. Select **Debug → Processor Events...** to open the List Selection window.
2. Select the processor events that you want enabled.

Note

Although vector catches set in this way override those set using the other methods, the settings are not retained after you disconnect from the target. To make permanent changes to settings, use one of the other methods described in this section.

4.11.5 Displaying variables

To display the value of a variable from your source code:

1. Connect to your debug target, if it is not already connected (see *Connecting to your target* on page 4-8).
2. Load the dhrystone.axf image as described in *Loading an image ready for debugging* on page 4-10. The source file `dhry_1.c` is loaded into the File Editor pane. If line numbers are not visible for your source, display them as described in *Displaying line numbers* on page 4-15.
3. Click the **Go** button on the toolbar to execute the image.
4. Enter the require number of runs, for example `50000`.
5. Select the required variable in the current context, for example click on the file tab for the source file `dhry_1.c` and move to line 301. Highlight the variable `Ptr_Glob` in the expression:

   ```
   structassign (*Ptr_Val_Par->Ptr_Comp, *Ptr_Glob);
   ```
6. Right-click to display the **Source Variable Name** menu, shown in Figure 4-1 on page 4-20.

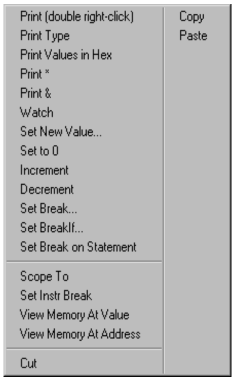


Figure 4-1 Source Variable Name menu

7. Select **Print** to view the value of the chosen variable in the current context. This is displayed in the **Cmd** tab of the Output pane.

8. Select **View Memory At Value** to display the memory view at this location.

4.12 Building and rebuilding an image with RealView Debugger

After you are familiar with the basic tasks for loading and debugging an image with RealView Debugger, the next step is to decide how you want to build your own image. You can:

- use RealView Debugger
- use the RVCT build tools directly. See *Getting started with the compilation tools* on page 4-27.

The following procedure describes how to rebuild the image for the RealView Debugger Dhrystone project. It assumes that you are familiar with the RealView Debugger desktop described in Chapter 3 *RealView Debugger Desktop*.

1. Before you rebuild the Dhrystone project, make a back up copy of the RealView Debugger `*examples_directory*\dhrystone` project directory.
2. Start RealView Debugger (see *Starting RealView Debugger* on page 4-5).
3. Open the RVDK Dhrystone example project file (`dhrystone.prj`) located in the RealView Debugger examples directory:

   `*examples_directory*\dhrystone`

   See *Opening an existing RealView Debugger project* on page 4-6 for instructions on how to open a project.
4. To rebuild the `dhrystone.axf` image, select the menu option **Build** → **Rebuild All**.

   If you see a prompt stating that the makefile does not exist, click **Yes** to build the makefile. RealView Debugger creates a makefile and a build directory for each of the build target configurations defined in the project (see *Build target configurations for Standard and Library projects* on page 4-23 for details)

   Build messages appear in the **Build** tab of the Output pane.
5. When the build is complete, you can load the image as described in *Loading an image associated with a RealView Debugger project* on page 4-10.

See the chapter on managing projects in the *RealView Developer Kit v2.2 Debugger User Guide* for more details on building your application.

4.13 Help on creating RealView Debugger projects

This section helps you to determine what kind of RealView Debugger project to create. It contains the following sections:

- *Types of project*
- *Project properties* on page 4-23
- *Limitations of Standard and Library projects* on page 4-24
- *Making a project and a connection interdependent* on page 4-25
- *Automatic operations performed by a project* on page 4-25.

For instructions on how to create a project, see the chapter on managing projects in the *RealView Developer Kit v2.2 Debugger User Guide*.

4.13.1 Types of project

In RealView Debugger, you can create the following types of project:

Standard Builds an executable image using the compiler, assembler and linker as appropriate. You add the sources used to build the image to the project, and RealView Debugger creates the necessary makefiles for you. However, there are limitation with this type of project (see *Limitations of Standard and Library projects* on page 4-24).

Library Builds an object library using the compiler, assembler, and the ARM® librarian utility as appropriate. You add the sources used to build the library to the project, and RealView Debugger creates the necessary makefiles for you. However, there are limitation with this type of project (see *Limitations of Standard and Library projects* on page 4-24).

Custom Builds any build target (executable image, ROM image, or object library) from your own makefile. You specify the location of your makefile, and any arguments that it takes. This type or project also enables you to overcome the limitations of the Standard and Library projects (see *Limitations of Standard and Library projects* on page 4-24).

Container Builds your application from subprojects you have previously created. You can add any combination of Standard, Library, and Custom subprojects. However, the order in which you add subprojects determines the build order.

4.13.2 Project properties

When you create a project, RealView Debugger sets up default project settings, called Project Properties. These settings include project information and build options that you can modify. The project properties are stored in project files that have the `.prj` file extension. A project file is located in the directory you specify when you create a project.

You can access the Project Properties for your project using a Project Properties window. For more details on accessing the Project Properties, and how to modify them, see the chapter on managing projects in the *RealView Developer Kit v2.2 Debugger User Guide*.

For more information about the compilation tool options used to implement the project build options, see *Getting started with the compilation tools* on page 4-27 and the compilation tools documentation.

Build target configurations for Standard and Library projects

When you create a Standard or Library project, RealView Debugger automatically opens a Project Properties window, and sets up default build target configurations, `Debug`, `DebugRel` and `Release`. You can create your own build target configurations as required. Each build target configuration can have different values for the build settings, but only one configuration is active for a project. For example, you might want to set up and use the `Debug` and `Release` configurations to build images for debugging and final release.

The first time you close the Project Properties window after creating a Standard or Library project, RealView Debugger creates a build directory and makefile for each build target configuration defined for the project.

——— **Note** ———

If a makefile and directory for the active build target configuration does not already exist when you build a project, RealView Debugger displays a prompt stating that the makefile does not exist. Click **Yes** at the prompt dialog. RealView Debugger creates the makefile and directory for the active build target configuration, then performs the build.

————————

See the chapter on managing projects in the *RealView Developer Kit v2.2 Debugger User Guide* for more details on build target configurations.

Preprocessing and post-processing commands

Standard and Library projects also enable you to specify other preprocessing and post-processing commands. For example, after RealView Debugger has built an image, you can set up a `fromelf` command that the project build process is to use to convert the image to a binary ROM image.

4.13.3 Limitations of Standard and Library projects

Table 4-3 describes the limitations with Standard and Library projects, and the consequences. To overcome these limitations, create your own makefile, and create a Custom project to use that makefile.

Table 4-3 Limitations of Standard and Library projects

Limitation	Consequence
Each source file you add is built with a separate assembler or compiler command.	You cannot have multiple source files in a single assembler or compiler command.
All files of the same language type (C, C++, or assembler, and the ARM and Thumb instruction set variants) are built using the same assembler or compiler options.	You cannot specify different compiler options for different source files of the same language type and instruction set variant. For example, if you have multiple source files written in C targeted at the ARM instruction set, they are all built using the same compiler options.

4.13.4 Making a project and a connection interdependent

RealView Debugger provides a mechanism that enables a project and a connection to be mutually dependent. This mutual dependence is controlled using the settings described in Table 4-4.

Table 4-4 Settings that control project and connection interdependence

Level of control	Setting	Description
Connection	`Project`	This is a setting that you specify for a connection. It identifies one or more projects that are to be opened automatically when you connect to a target processor on that connection. For more details, see the appendix describing the connection properties in the *RealView Developer Kit v2.2 Debugger User Guide*.
Project	`Specific_device`	This is a setting that you specify for a project. It identifies a specific processor (for example, ARM946E-S) or processor family (for example, ARM7™). This restricts the loading of the image for this project to a connection with the specified processor or processor family. RealView Debugger acts on this setting only when: • a connection to a matching target processor already exists, and you open the project • the project is already open, and you connect to a matching target processor. This close coupling of project and target processor is referred to as *specific device binding* or *autobinding*. It is particularly useful if you are working with multiple projects open. For more details, see the chapter on managing projects in the *RealView Developer Kit v2.2 Debugger User Guide*. Other project-related settings are provided that enable you to control what happens when a project interacts with a connection (see *Automatic operations performed by a project*).

4.13.5 Automatic operations performed by a project

A RealView Debugger project enables various operations to be performed automatically. To enable these operations to be performed automatically, RealView Debugger associates the project with a connection using a mechanism called *binding*. However, RealView Debugger only binds a project to a connection in the following circumstances:

- when a connection already exists, and the project you open is the first one opened in the current debugging session
- when one or more projects are open, and you connect to a target processor

- when you open an autobound project, that is a project associated with a specific device (see Table 4-4 on page 4-25), and a connection has a target processor that matches the device.

If you are working with multiple projects, there are specific rules that RealView Debugger uses to determine which project to bind to a connection. This might involve displaying a prompt to which you must respond.

If you do not restrict image loading to a specific device (see Table 4-4 on page 4-25), then the binding mechanism is called *default binding*.

In addition to invoking these automatic operations, project binding has other effects when you are working with projects. See the chapter on managing projects in the *RealView Developer Kit v2.2 Debugger User Guide* for full details on project binding.

Setting up the automatic operations for a project

You set up the operations in the Project Properties (see *Project properties* on page 4-23). Table 4-5 lists these operations and shows you where to find the information to implement them.

Table 4-5 Project-related operations

Operation	Reference
Load the image associated with a project, if it exists.	See the description of the `Open_load` setting in the *RealView Developer Kit v2.2 Debugger User Guide*
Set the initial load state of the image, which is one of: • register image name only • load symbols and image • load symbols only.	See the description of the `Open_load` setting in the *RealView Developer Kit v2.2 Debugger User Guide*
Set any image-related controls. For example, set the program counter (PC) to the image entry point.	See the descriptions of the settings in the `Image_load` group in the *RealView Developer Kit v2.2 Debugger User Guide*
Set various runtime controls, such as top of memory, and command-line arguments if the image accepts these.	See the descriptions of the settings in the `Runtime_Control` group in the *RealView Developer Kit v2.2 Debugger User Guide*
Set one or more predefined breakpoints.	See the descriptions of the settings in the `Auto_Set_Breaks` and `Named_Breaks` groups in the *RealView Developer Kit v2.2 Debugger User Guide*
Run one or more RealView Debugger CLI commands.	See the descriptions of the settings in the `Command_Open_Close` group in the *RealView Developer Kit v2.2 Debugger User Guide*

4.14 Getting started with the compilation tools

This section describes how to use the compilation tools at the command prompt. It describes the basic tasks you are most likely to perform with the compiler, assembler, linker, the fromELF utility, and ARM librarian utility. It includes the following sections:

- *Targeting the source language with the compiler* on page 4-28
- *Targeting the ARM or Thumb instruction set* on page 4-29
- *Targeting a specific ARM architecture or processor* on page 4-30
- *Targeting specific procedure call standard variants* on page 4-30
- *Generating debug information* on page 4-30
- *Optimizing your compiled sources* on page 4-30
- *Building an image with a single compiler invocation* on page 4-31
- *Building an image with separate command invocations* on page 4-31
- *Specifying the initial entry point for an image* on page 4-31
- *Creating object libraries* on page 4-32
- *Converting images to binary files* on page 4-32
- *Creating an image memory map with scatter-loading* on page 4-32
- *Building the RVDK example Dhrystone project* on page 4-33.

The compilation tools are described in detail in the following documents:

- *RealView Developer Kit v2.2 Assembler Guide*
- *RealView Developer Kit v2.2 Compiler and Libraries Guide*
- *RealView Developer Kit v2.2 Linker and Utilities Guide*

Note

If you create a RealView Debugger project, you can set up the build tool options using the RealView Debugger GUI. See the chapter on managing projects in the *RealView Developer Kit v2.2 Debugger User Guide*. Also, see *Help on creating RealView Debugger projects* on page 4-22, which provides information to help you decide the type of RealView Debugger project to create.

4.14.1 Targeting the source language with the compiler

To target a specific source language with the compiler:

ISO standard C

Use the compiler option:

```
armcc --c90
```

This is the default option.

Strict ISO standard C

Use the compiler options:

```
armcc --c90 --strict
```

ISO standard C++

Use the compiler option:

```
armcc --cpp
```

Strict ISO standard C++

Use the compiler options:

```
armcc --cpp --strict
```

If you do not specify any of these options, the compiler determines the source language from the extension of your source files. However, if you specify multiple source files for a single compiler invocation that contain different source languages, then specify one of these compiler options. For example, to compile `f1.c` and `f2.cpp` for ISO standard C++:

```
armcc --cpp f1.c f2.cpp
```

If you specify any assembly language sources, then the compiler invokes the assembler to assemble these source files.

You can also use the `armcpp` command to invoke the C++ compiler.

4.14.2 Targeting the ARM or Thumb instruction set

You can compile or assemble your sources for the ARM® or Thumb instruction set. Table 4-6 shows the compiler and assembler functionality that enables you to do this.

Table 4-6 Targeting the ARM and Thumb instruction sets

Target instruction set	Compiler	Assembler	Scope
ARM	`armcc --arm`	`armasm -32` or `--arm`	source file(s)
ARM	`#pragma arm`	`CODE32` directive	per-function[a]
Thumb	`armcc --thumb`	`armasm -16` or `--thumb`	source file(s)
Thumb	`#pragma thumb`	`CODE16` directive	per-function[a]

a. If your source code contains these instructions, they override the opposing command-line options. For example, the `CODE16` directive overrides the command `armasm -32`.

If you include the pragmas or directives in your source code, you generate mixed instruction set code. For this to work properly, you must also use the interworking compiler option. See *Targeting specific procedure call standard variants* on page 4-30 for details.

To see how the assembler directives are used, examine the RVDK example assembler file `*examples_directory*\asm\thumbsub.s`.

You can also use alternative compiler commands shown in Table 4-7 to compile your C and C++ code for ARM and Thumb instructions sets.

Table 4-7 Alternative compiler commands

Command	Description	Equivalent armcc command
`armcc`	Compiles for ISO standard C, and the ARM instruction set, but only if your source files have `.c` extensions	`armcc`
`armcpp`	Compiles for ISO standard C++, and the ARM instruction set	`armcc --cpp`
`tcc`	Compiles for ISO standard C, and the Thumb instruction set	`armcc --thumb`
`tcpp`	Compiles for ISO standard C++, and the Thumb instruction set	`armcc --thumb --cpp`

4.14.3 Targeting a specific ARM architecture or processor

To compile or assemble your sources for a specific ARM architecture or processor, use the `--cpu` option. This option is the same for the compiler and assembler. For example:

`--cpu 5TE` Targets ARM architecture v5TE.

`--cpu ARM946E-S`

Targets the ARM946E-S processor.

For more details about the `--cpu` option, see the *RealView Developer Kit v2.2 Assembler Guide* and the *RealView Developer Kit v2.2 Compiler and Libraries Guide*.

4.14.4 Targeting specific procedure call standard variants

You can specify the following procedure call standard variant.

Interworking

If your application is built from sources that target both the ARM and Thumb instruction sets, you must specify interworking:

```
--apcs /interwork
```

This option is the same in both the compiler and assembler.

Examine the RVDK example project `*examples_directory*\interwork` to see how this option is used.

For more details about this, and other `--apcs` options, see the *RealView Developer Kit v2.2 Assembler Guide* and the *RealView Developer Kit v2.2 Compiler and Libraries Guide*.

4.14.5 Generating debug information

If you want to debug your image using RealView Debugger, specify the `-g` or `--debug` option. This option name is the same in both the compiler and assembler.

By default, the linker includes debug information (see the description of the `--debug` option in the *RealView Developer Kit v2.2 Linker and Utilities Guide*).

4.14.6 Optimizing your compiled sources

When you are compiling C or C++ code, the compiler enables you to optimize the generated code using optimization levels O1, O2 and O3. The default option is O2.

For more details on the optimizations that are available, see the *RealView Developer Kit v2.2 Compiler and Libraries Guide*.

4.14.7 Building an image with a single compiler invocation

By default, the compiler attempts to generate an image from the specified source files. The default image name is `__image.axf`.

To specify your own image name, use the `-o` *`image_name`* option. For example:

```
armcc f1.c f2.c -o myimage.axf
```

For more details, see the *RealView Developer Kit v2.2 Compiler and Libraries Guide*.

4.14.8 Building an image with separate command invocations

To build an image with separate compiler, assembler, and linker commands:

1. For C and C++ sources, specify the option `-c` to force the compiler to compile only, and not link the generated object files. For example:

   ```
   armcc -c f1.c f2.c
   ```

   For more details, see the *RealView Developer Kit v2.2 Compiler and Libraries Guide*.

   Note

   For assembler sources, the assembler (`armasm`) only generates object files.

2. Specify the object files generated in step 1 as part of the linker command. By default, the linker generates an image with the name `__image.axf`.

   To specify your own image name, use the `--output` *`image_name`* option. For example:

   ```
   armlink f1.o f2.o --output myimage.axf
   ```

   For more details, see the *RealView Developer Kit v2.2 Linker and Utilities Guide*.

4.14.9 Specifying the initial entry point for an image

If your image contains multiple entry points, you must specify a unique initial entry point for the image. Use the `--entry` linker option. For example:

```
armlink --entry 0x8000
```

For more details, see the *RealView Developer Kit v2.2 Linker and Utilities Guide*.

4.14.10 Creating object libraries

The ARM librarian utility enables you to create object libraries. For example, to create a library called `mylib` containing all object files in the current directory, enter the command:

```
armar -create mylib *.o
```

For more information on using the ARM librarian, see the *RealView Developer Kit v2.2 Linker and Utilities Guide*.

4.14.11 Converting images to binary files

The fromELF utility enables you to convert your images to different formats. For example, to convert the image `infile.axf` to a plain binary file `outfile.bin` for downloading to flash, use the command:

```
fromelf -bin -o outfile.bin infile.axf
```

For more information on using the fromELF utility, see the *RealView Developer Kit v2.2 Linker and Utilities Guide*.

4.14.12 Creating an image memory map with scatter-loading

To create an image memory map, link using a scatter-load description file:

```
armlink --scatter file
```

Examine the RVDK example project *`examples_directory`*`\emb_sw_dev` to see an example scatter file and how it is used.

For more details on creating and using scatter-load description files, see the *RealView Developer Kit v2.2 Linker and Utilities Guide*.

4.14.13 Building the RVDK example Dhrystone project

You can build the RVDK example Dhrystone project using the project makefile (`dhry.mk`) or the Windows command file (`dhry.bat`).

The following commands and options are used to build this project:

```
armcc -c -W -g -O2 -Otime -Ono_inline -DMSC_CLOCK dhry_1.c dhry_2.c
armlink dhry_1.o dhry_2.o -o dhry.axf --info totals
```

Table 4-8 describes the options used in these commands.

Table 4-8 Command options used to build the dhry example project

Command option	Description
`-c`	Instructs the compiler to create the object files only.
`-W`	Instructs the compiler to suppress all warning messages.
`-g`	Instructs the compiler to include debug tables.
`-O2`	Instructs the compiler to generate fully optimized code.
`-Otime`	Instructs the compiler to perform optimizations to reduce execution time at the possible expense of a larger image.
`-Ono_inline`	Instructs the compiler to disable the inlining of functions.
`-DMSC_CLOCK`	Instructs the compiler to define the symbol `MSC_CLOCK` as a preprocessor macro.
`-o dhry.axf`	Instructs the linker to create an image with the filename `dhry.axf`.
`--info totals`	Instructs the linker to display the totals of the Code and Data (RO Data, RW Data, ZI Data, and Debug Data) sizes for input objects and libraries.

See the *RealView Developer Kit v2.2 Compiler and Libraries Guide* for a description of the compiler options.

See the *RealView Developer Kit v2.2 Linker and Utilities Guide* for a description of the linker options.

Appendix A
Using the armenv Tool

This appendix describes the `armenv` tool that you can use to manage your ARM RealView product installations. It includes the following sections:

- *About the armenv tool* on page A-2
- *Using the armenv tool* on page A-3.

A.1 About the armenv tool

The `armenv` tool enables you to:

- set up and remove the environment variables for ARM RealView products
- check for clashes between the ARM RealView products you have installed
- set up different varieties of the same product.

Note

You cannot use the `armenv` tool for Custom installations in this release of RVDK.

You can find the `armenv` tool at:

`install_directory/bin/platform`

A.2 Using the armenv tool

This section describes the syntax of the `armenv` command, and shows some examples of how it can be used. It includes:

- *armenv command syntax*
- *armenv command-line arguments*

A.2.1 armenv command syntax

The command syntax of the `armenv` tool is:

```
armenv [-r root] [-u] -p product [[--and] -p product]... [--user|--sys|--proc]
[--bat|--sh|--csh|--posh|--exec program [args]]
```

The arguments are described in *armenv command-line arguments*.

A.2.2 armenv command-line arguments

Table A-1 shows the command-line arguments that are available.

Table A-1 Generic armenv arguments

Argument	Description
`--help`	Displays help on the command-line arguments.
`-r` *`root`*	The absolute path to the root of the product installation, *`install_directory`*. For example, on Windows the default root is: `C:\Program Files\ARM`
`-p` *`product`*	The ARM RealView product. See *Product syntax* on page A-4 for more details.
`--and`	Compute settings for all products before this argument, then do the same for those following it. The settings in the second group override those in the first.

Table A-1 Generic armenv arguments (continued)

Argument	Description
`--proc`	Change the environment for the current process only. You cannot use this argument with `--system` or `--user`.
`--exec`	Execute a program in the new environment. You cannot use this argument with `--bat`.
`-u`	Attempts to undo the changes to the environment that were made when setting up the product.

Table A-2 shows the command-line arguments that are specific to Windows systems.

Table A-2 armenv arguments specific to Windows

Argument	Description
`--system`	Update the Windows SYSTEM area of the registry. This is the default.
`--user`	Update the Windows USER area of the registry.
`--bat`	Changes the environment for the current command prompt window. This is the default.

Product syntax

The syntax for specifying the product is:

```
-p category [name] [version [rev]] [-v name value]...
```

where:

category	The product identifier, for example, `RVDK`.
name	Do not use this argument (`armenv` uses the default name `Contents`).
version	The version number of the product, for example, `2.2`. If you do not specify a version, the most recent version of the installed product is used.
rev	A specific build number for the product. If you do not specify a build number, the most recent build of the installed product is used.

`-v` *`name value`*

Identifies a variant of the same product.

`name`	The type of the variant, for example, `platform`. It is suggested that you use only the `platform` variant.
`value`	The specific variant, for example, `win_32-pentium`.

For example: `-v platform win_32-pentium`.

Glossary

American National Standards Institute (ANSI)	An organization that specifies standards for, among other things, computer software. This is superseded by the International Standards Organization.
APCS	ARM Procedure Call Standard.
ARM instruction	A word that specifies an operation for an ARM processor to perform. ARM instructions must be word-aligned.
ARM Toolkit Proprietary ELF (ATPE)	The binary file format used by RealView Developer Kit. ATPE object format is produced by the compiler and assembler tools. The ARM linker accepts ATPE object files and can output an ATPE executable file. RealView Debugger can load only ATPE format images, or binary ROM images produced by the fromELF utility. *See also* RealView Developer Suite.
armasm	The ARM assembler.
armcc	The ARM C compiler.
ARM-Thumb Procedure Call Standard (ATPCS)	Defines how registers and the stack are used for subroutine calls.
ATPCS	*See* ARM-Thumb Procedure Call Standard.
ATPE	*See* ARM Toolkit Proprietary ELF.

Breakpoint	A location in the image. If execution reaches this location, the debugger halts execution of the image.
C file	A file containing C source code.
Cache	A block of high-speed memory locations whose addresses are changed automatically in response to those memory locations the processor is accessing, and whose purpose is to increase the average speed of a memory access.
CLI	C Language Interface/Command-Line Interface.
Command-line Interface	You can operate RealView Debugger by issuing commands in response to command-line prompts. A command-line interface is particularly useful when you have to run the same sequence of commands repeatedly. You can store the commands in a file and submit that file to the command-line interface of the debugger.
Compilation	The process of converting a high-level language (such as C or C++) into an object file.
CPU	Central Processor Unit.
C, C++	Programming languages.
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).
Debugger	An application that monitors and controls the execution of a second application. Usually used to find errors in the application program flow.
DWARF	*See* Debug With Arbitrary Record Format.
Embedded	Applications that are developed as firmware. Assembler functions placed out-of-line in a C or C++ program. *See also* Inline.
GUI	Graphical User Interface.
Host	A computer that provides data and other services to another computer.
IEEE	Institute of Electrical and Electronic Engineers (USA).
Image	An execution file that has been loaded onto a processor for execution.
Inline	Functions that are repeated in code each time they are used rather than having a common subroutine. Assembler code placed within a C or C++ program. *See also* Embedded.

Input section	Contains code or initialized data or describes a fragment of memory that must be set to zero before the application starts.
Interworking	A method of working that allows branches between ARM and Thumb code.
International Standards Organization (ISO)	An organization that specifies standards for, among other things, computer software. This supersedes the American National Standards Institute.
ISO	*See* International Standards Organization.
Library	A collection of assembler or compiler output objects grouped together into a single repository.
Linker	Software that produces a single image from one or more source assembler or compiler output objects.
Output section	A contiguous sequence of input sections that have the same RO, RW, or ZI attributes. The sections are grouped together in larger fragments called regions. The regions are grouped together into the final executable image.
PC	*See* Program Counter.
Processor	An actual processor, real or emulated running on the target. A processor always has at least one context of execution.
Program Counter (PC)	Integer register R15.
RAM	Random Access Memory.
RealView ICE (RVI)	A JTAG-based debug solution to debug software running on ARM processors.
RealView ICE Micro Edition (RVI-ME)	A JTAG-based debug tool for embedded systems.
RealView Trace	Provides tracing functionality for RealView ICE.
Regions	A contiguous sequence of one to three output sections (RO, RW, and ZI) in an image.
Register	A processor register.
RISC	Reduced Instruction Set Computer.
ROM	Read Only Memory.
RTOS	Real-Time Operating System.
RVI	*See* RealView ICE.
RVI-ME	*See* RealView ICE Micro Edition.

RVT	*See* RealView Trace.
Scatter loading	Assigning the address and grouping of code and data sections individually rather than using single large blocks.
Scope	The accessibility of a function or variable at a particular point in the application code. Symbols that have global scope are always accessible. Symbols with local or private scope are only accessible to code in the same subroutine or object.
Source File	A file that is processed as part of the image building process. Source files are associated with images.
Target	The actual target processor, (real or simulated), on which the application is running.
	The fundamental object in any debugging session. The basis of the debugging system. The environment in which the target software will run. It is essentially a collection of real or simulated processors.
TCC	Thumb C Compiler.
Thumb instruction	A halfword that specifies an operation for an ARM processor in Thumb state to perform. Thumb instructions must be halfword-aligned.
Variable	A named memory location of an appropriate size to hold a specific data item.

Index