RealView® Developer Kit for ST

Version 1.0

Getting Started Guide

ARM®

ARM DUI 0270A

RealView Developer Kit for ST
Getting Started Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
March 2004	A	RVDK for ST v1.0 Release

Proprietary Notice

Confidentiality Status

This document is Open Access. This document has no restriction on distribution.

Product Status

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

Web Address

`http://www.arm.com`

Contents
RealView Developer Kit for ST Getting Started Guide

Preface
About this book vi
Feedback viii

Chapter 1 Introduction
1.1 RealView Developer Kit components 1-2
1.2 RealView Developer Kit documentation 1-6

Chapter 2 Features of the Compilation Tools, the Debugger, and Debug Target Hardware
2.1 ARM Toolkit Proprietary ELF format 2-2
2.2 RealView Compilation Tools v2.0.1 2-3
2.3 RealView Debugger v1.6.1 2-6
2.4 RealView ICE Micro Edition v1.1 2-9

Chapter 3 RealView Debugger Desktop
3.1 Basic elements of the desktop 3-2
3.2 Finding options on the main menu 3-11
3.3 Working with toolbars 3-13
3.4 Working in the Code window 3-16

3.5 Connection Control window 3-19
3.6 Editor window 3-20
3.7 Resource Viewer window 3-21

Chapter 4 Getting Started with RealView Developer Kit
4.1 Building and debugging task overview 4-2
4.2 Using the example projects 4-4
4.3 Starting and Exiting RealView Debugger 4-6
4.4 Opening an existing RealView Debugger project 4-7
4.5 Changing your target board/chip definition 4-8
4.6 Connecting to a debug target 4-9
4.7 Loading an image ready for debugging 4-10
4.8 Unloading an image 4-12
4.9 Running the image 4-13
4.10 Basic debugging tasks with RealView Debugger 4-14
4.11 Building and rebuilding an image with RealView Debugger 4-17
4.12 Help on creating RealView Debugger projects 4-18
4.13 Getting started with the compilation tools 4-23

Glossary

Preface

This preface introduces the *RealView® Developer Kit for ST v1.0 Getting Started Guide*. 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 for ST* (RVDK) v1.0 tools and documentation.

Intended audience

This book is written for all developers who are producing applications using RVDK. It assumes that you are an experienced software developer, but are not 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 v1.0.

Chapter 2 ***Features of the Compilation Tools, the Debugger, and Debug Target Hardware***

Read this chapter for a description of the major features of the RVDK v1.0 compilation tools, debugger, and RealView ICE Micro Edition. The online documentation is also described.

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***

This chapter gives a high-level summary of the tasks to begin using the RVDK debugger and compilation tools.

Typographical conventions

The following typographical conventions are used in this book:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes ARM processor signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
`monospace`	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
`monospace italic`	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.

Further reading

This section lists publications from 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 v1.0 Assembler Guide* (ARM DUI 0231)
- *RealView Developer Kit v1.0 Compiler and Libraries Guide* (ARM DUI 0232)
- *RealView Developer Kit v1.0 Linker and Utilities Guide* (ARM DUI 0233)
- *RealView Developer Kit v1.0 Debugger User Guide* (ARM DUI 0234)
- *RealView Developer Kit v1.0 Debugger Command Line Reference Guide* (ARM DUI 0235)
- *RealView ICE Micro Edition v1.1 User Guide* (ARM DUI 0220).

For general information on software interfaces and standards supported by the ARM tools, see `http://www.arm.com`.

See the following documentation for specific information relating to ARM products:

- *ARM Architecture Reference Manual*
- *ARM Reference Peripheral Specification* (ARM DDI 0062).

Also, see the ARM technical reference manuals for the supported ARM cores.

Feedback

ARM Limited welcomes feedback on both RealView Developer Kit for ST and its documentation.

Feedback on RealView Developer Kit for ST

If you have any problems with RealView Developer Kit for ST, 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 for ST v1.0* (RVDK v1.0) and describes the component products and documentation. It contains the following sections:

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

1.1 RealView Developer Kit components

RVDK v1.0 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.

Table 1-1 lists the components of RVDK v1.0, 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 v2.0.1*
Debugger	*RealView Debugger v1.6.1* on page 1-3
Debug target hardware	*RealView ICE Micro Edition v1.1* on page 1-4

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 4-4).

1.1.1 RealView Compilation Tools v2.0.1

You can use *RealView Compilation Tools v2.0.1* to build C, C++, or ARM assembly language programs. See *Getting started with the compilation tools* on page 4-23 for an introduction on using the compilation tools.

Note

By default, the compiler enables you to compile C code only. However, the RVDK documentation includes descriptions of features that are specific to C++. To compile C++ code, you must purchase the RVDK C++ upgrade.

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*version_num**build_num*\st\win_32-pentium
includes_directory	*install_directory*\RVCT\Data\2.0.1*build_num*\include\windows
examples_directory	*install_directory*\RVCT\Examples\2.0.1\release\windows

1.1.2 RealView Debugger v1.6.1

RealView Debugger v1.6.1, 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\1.6.1*build_num*\st\win_32-pentium
examples_directory	*install_directory*\RVD\Examples\1.6.1\release_2\windows

1.1.3 RealView ICE Micro Edition v1.1

RealView ICE Micro Edition v1.1, together with your target board and RealView Debugger v1.6.1, enables you to develop and debug your embedded applications.

1.1.4 RVDK example projects

Example projects are provided with RVDK, and includes:

- Projects that are set up specifically for the boards and processors supported by RVDK for ST. These are located in the directory:

  install_directory\ARM\RVMDK\Examples\1.0.1\17\st\windows

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

  Programs → ARM → ARM RealView Developer Kit for ST v1.0 → Examples → ***project_name***

  Note

  If you are already using RealView Debugger, you must open the project using RealView Debugger (see *Opening an existing RealView Debugger project* on page 4-7). This is because selecting the project from the **Start** menu also starts RealView Debugger.

  To build these projects, see *Building and rebuilding an image with RealView Debugger* on page 4-17.

- RVCT examples are located in the directory shown in Table 1-2 on page 1-3. You can also access these examples from the Windows **Start** menu:

  Programs → ARM → ARM RealView Developer Kit for ST v1.0 → Examples → RVCT Examples

- RealView Debugger examples are located in the directory shown in Table 1-3 on page 1-3. You can also access these examples from the Windows **Start** menu:

  Programs → ARM → ARM RealView Developer Kit for ST v1.0 → Examples → RVD Examples

1.1.5 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-ME` is used as an example access-provider connection
- `MCUeval` is used as an example board file name
- `ARM7TDMI` is used as an example chip name
- `ARM7TDMI_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 documentation

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

1.2.1 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 → ARM RealView Developer Kit for ST v1.0 → PDF Documentation

The PDF files are installed in the following directories:

- RealView Developer Kit v1.0 compilation tools and debugger documentation:

  install_directory\Documentation\RVDKST\1.0\release\windows\PDF

  install_directory\Documentation\RVDK\1.1\release\windows\PDF

- RealView ICE Micro Edition v1.1 documentation:

  install_directory\Documentation\RVDK\1.1\release\windows\PDF.

Rogue Wave C++ library manuals

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, and Debug Target Hardware

This chapter describes the features of the RealView® Developer Kit (RVDK) compilation tools, the debugger, and the debug target hardware. It contains the following sections:

- *ARM Toolkit Proprietary ELF format* on page 2-2
- *RealView Compilation Tools v2.0.1* on page 2-3
- *RealView Debugger v1.6.1* on page 2-6
- *RealView ICE Micro Edition v1.1* on page 2-9.

2.1 ARM Toolkit Proprietary ELF format

The *ARM® Toolkit Proprietary ELF* (ATPE) format is supported only by the RVDK for ST 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 ST.

2.2 RealView Compilation Tools v2.0.1

This section describes the *RealView Compilation Tools v2.0.1* (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.

Note

By default, the RVDK v1.0 compiler enables you to compile C code only. However, the RVDK documentation includes descriptions of features that are specific to C++. To compile C++ code, you must purchase the RVDK C++ upgrade.

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.

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.

C++ The ARM compiler supports the full ISO C++ language, except for exceptions.

ABI for the ARM Architecture

The *Application Binary Interface for the ARM Architecture* (ABI for the ARM Architecture) enables you to use the ARM and Thumb objects and libraries from different producers that support the ABI for the ARM Architecture. For more details, see `http://www.arm.com`.

Supporting software

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

2.3 RealView Debugger v1.6.1

This section describes the features available in RealView Debugger v1.6.1. 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 only ATPE format images generated by `armlink`, or flash/ROM images generated by `fromELF`.

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 v1.6.1 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 v1.6.1 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 v1.6.1 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 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 v1.0 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 v1.0 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.

Chapter 3
RealView Debugger Desktop

This chapter describes, in detail, the basic elements of the RealView® Debugger desktop. It contains the following sections:

- *Basic elements of the desktop* on page 3-2
- *Finding options on the main menu* on page 3-11
- *Working with toolbars* on page 3-13
- *Working in the Code window* on page 3-16
- *Connection Control window* on page 3-19
- *Editor window* on page 3-20
- *Resource Viewer window* on page 3-21.

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. It contains the following sections:

- *Splash screen*
- *Code window*
- *Default windows and panes* on page 3-5
- *Pane controls* on page 3-7
- *Toolbars* on page 3-8
- *Color Box* on page 3-9
- *Other window elements* on page 3-9.

3.1.1 Splash screen

When you start RealView Debugger, the RealView Debugger splash screen is displayed. During this time, the debugger is checking your environment and creating (or updating) configuration files and a working directory.

Use the `-nologo` flag to run RealView Debugger from the command line without a splash screen.

3.1.2 Code window

The Windows splash screen is replaced by the default 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 on page 3-3.

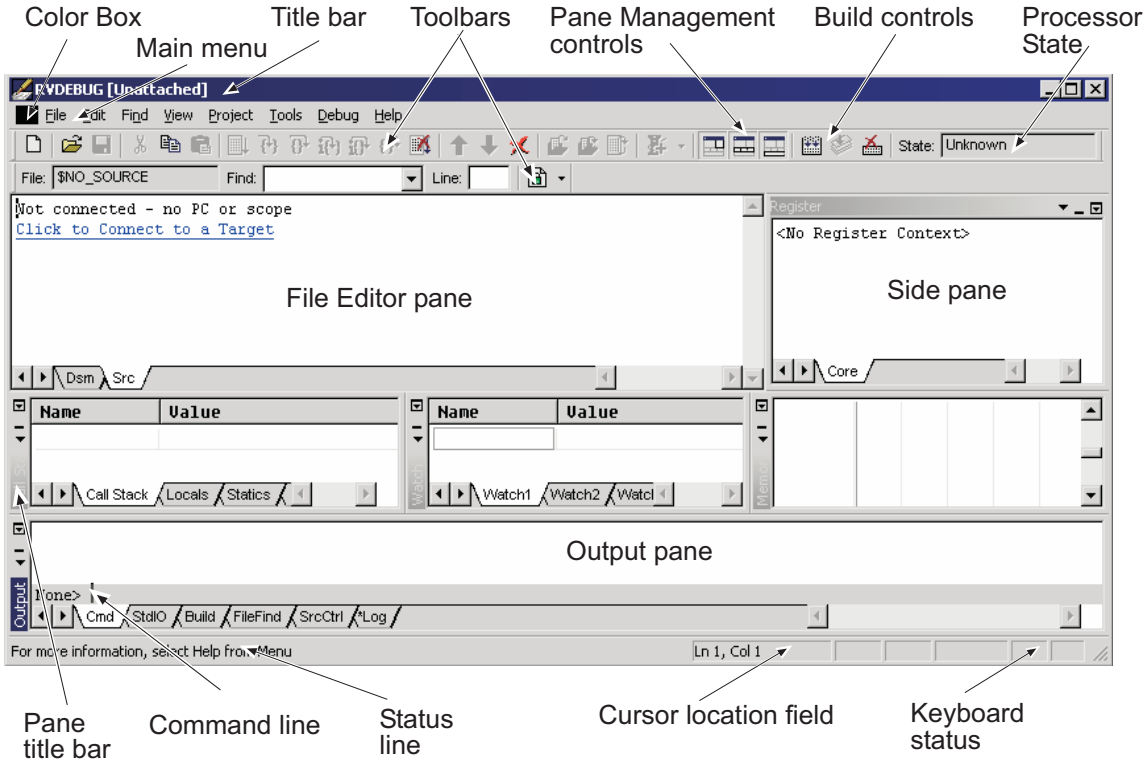


Figure 3-1 Default Code window

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

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

The toolbar buttons displayed, and the options available from window and pane menus also change depending on the licenses you have.

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`.

(dhrystone) 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, `(my_project)`.

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 describing managing projects in *RealView Developer Kit v1.0 Debugger User Guide* for details on project binding.

Also, see *Automatic operations performed by a project* on page 4-21.

***@target_processor*:ARM-ARM-USB**

The connection, including the target processor (for example, `ARM7TDMI_0`) and the execution vehicle, `ARM-ARM-USB` (see *Connection Control window* on page 3-19 for more details).

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

```
T0x19F84_ICTM.ARM
```

See the chapter describing RTOS support in *RealView Developer Kit v1.0 Debugger User Guide* for details.

[Unattached]

The attachment of the window to a specified thread.

If you are using an RTOS, you can 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 describing RTOS support in *RealView Developer Kit v1.0 Debugger User Guide* for details.

If you float a pane, the pane title bar reflects the title bar of the calling Code window.

3.1.3 Default windows and panes

RealView Debugger provides a range of debug views:

- Registers
- Stack
- Process Control
- Symbol Browser
- Watch
- Call Stack and Locals
- Memory and Memory Map
- Threads (with the appropriate RTOS support package)
- Break/Tracepoints.

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

File Editor pane

The File Editor pane is always visible when working with RealView Debugger.

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.

Side pane By default, this contains the Register pane view that displays grouped processor registers for the current target processor. When you first run RealView Debugger, this pane is positioned to the right of the File Editor pane but you can switch this pane to the left, using the **Switch Side** option from the **Pane Content** menu.

Middle pane row

The middle pane row can contain one, two, or three pane views, but you can specify which panes are visible. By default this row contains:

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.

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

The bottom pane of the Code window always contains the Output pane. 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. This shows the status of the current process, for example, `Stop`, `Run`, or `None` (no process). You can also enter debugger commands at the `>` prompt.

During your debugging session, you can use the **Pane Content** menu to define which view is displayed in a chosen pane (see *Pane controls* for details). At the end of your first session, you can configure RealView Debugger to start next time with a different set of panes or views if required by saving the customized setup in your workspace.

Changing panes

Select **View** → **Pane Views** to change what is displayed in a pane view or to restore the default views. You can also use this menu to display any window or pane that is hidden, without changing the view it displays. If a pane is hidden and you use the **Pane Views** menu to change the pane contents, it is automatically displayed to show the new view.

Formatting pane views

The contents of the File Editor pane and the Output pane cannot be changed. However, you can define how the view is formatted, for example change the size of text displayed in the File Editor pane or the number of lines displayed in the Output pane. See the chapter describing configuring workspaces in *RealView Developer Kit v1.0 Debugger User Guide* for details of how to do this.

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.

If you float a pane, the title bar is displayed horizontally at the top of the window.

A pane contains the controls:

Pane Content

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

The selected option in the menu indicates the current view.

Visual controls

The visual controls are at the bottom of the **Pane Content** menu. Use these to float or hide the pane.

In the side pane, use the option **Switch Side** to move the pane from the right side of the Code window so that it is positioned to the left of the code view.

Expand/Collapse Pane

Click this button to collapse the currently selected view. Other panes are expanded to fill the empty area.

Click the **Expand Pane** button to restore the view.

There is no option to collapse, or expand, a floating pane.

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.

3.1.5 Toolbars

There are two toolbars, below the Code window main menu, that provide quick access to many of the features available from menu options:

- Actions toolbar, shown in Figure 3-2
- Editing toolbar, shown in Figure 3-3 on page 3-9.

Figure 3-2 Actions toolbar

Figure 3-3 Editing toolbar

To disable a toolbar, select **View** → **Toolbars** from the main menu. 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. To restore the floating toolbar, double-click anywhere on the toolbar title bar.

Note

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

See *Working with toolbars* on page 3-13 for details on the buttons available from the Actions toolbar and the Editing toolbar, and how to customize toolbar groups.

3.1.6 Color Box

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

When connected to your target processor, the Color Box identifies the connection associated with the window. This is located on the main menu toolbar next to **File**, shown in Figure 3-1 on page 3-3.

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. 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:

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.

Cursor location field

As you move through files within the File Editor pane, the current location of the text insertion point is displayed in the Cursor location field at the bottom right of the Code window.

LOG Shows that output is being written to a log file.

JOU Shows that output is being written to a journal file.

STDIOlog Shows that output is being written to an STDIOlog file.

NUM Shows that you can use the numeric keypad to enter numbers.

CAP Shows that Caps Lock is selected.

3.2 Finding options on the main menu

This section provides a summary of the main menu options available from the Code window that enable you to:

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

The options available from the main menu bar are:

File Enables you to:

- open and close files within the File Editor pane
- load images (see the chapter on working with image in the *RealView Developer Kit v1.0 Debugger User Guide*)
- connect to targets using the Connection Control window (see *Connection Control window* on page 3-19)
- manage target connections (see the chapters describing connecting and configuring targets in the *RealView Developer Kit v1.0 Debugger User Guide*)
- open and save a workspaces (see the chapter on configuring workspace settings in the *RealView Developer Kit v1.0 Debugger User Guide*).

Note

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 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 (see the chapter on editing source code in the *RealView Developer Kit v1.0 Debugger User Guide*).

Find Enables you to work with source files and to perform searches on those files as you debug your image (see the chapter on searching and replacing text in the *RealView Developer Kit v1.0 Debugger User Guide*).

This also gives access to the browsers in RealView Debugger (see the chapter on working with browsers in the *RealView Developer Kit v1.0 Debugger User Guide*).

View Enables you to:

- display the following windows during your debugging session
  - another Code window
  - a Resource Viewer window (see *Resource Viewer window* on page 3-21)
  - an Editor window (see *Editor window* on page 3-20).
- set up new panes during your debugging session (see *Default windows and panes* on page 3-5 and *Pane controls* on page 3-7).
- customize toolbar groups (see *Toolbars* on page 3-8).

Note

The **Custom Windows** option is not available in this release.

Project Enables you to work with projects so that you can organize your source files, model your build process, and share files with other developers (see the chapter on managing projects in the *RealView Developer Kit v1.0 Debugger User Guide*). This can be used in conjunction with the **Tools** menu to rebuild images.

Tools Enables you to build files, or groups of files, to create your image ready for loading to a target (see the chapter on managing projects in the *RealView Developer Kit v1.0 Debugger User Guide*).

It also enables you to examine, and change, your workspace settings and global configuration options (see the chapter on configuring workspace settings in the *RealView Developer Kit v1.0 Debugger User Guide*). Use the **File** menu to open and save a workspace.

Debug Includes the main facilities you use during a debugging session (see the chapters on controlling execution and working with breakpoints in the *RealView Developer Kit v1.0 Debugger User Guide*).

Help 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 toolbars

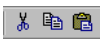
The Code window toolbars give access to many of the features available from the main menu and to additional debugging controls. By default, the Code window shows both toolbars but you can customize which controls are available. This section describes:

- *Actions toolbar*
- *Editing toolbar* on page 3-14
- *Customizing the Actions toolbar* on page 3-15.

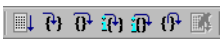
3.3.1 Actions toolbar

The Actions toolbar, see Figure 3-2 on page 3-8, contains buttons used during debugging sessions and when working with source code:

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



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



Execution group

Use these buttons to control program execution, for example starting and stopping execution, and stepping.

These buttons are enabled when an image has been loaded. They replicate selected options from the **Debug** menu.

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.

Image load group

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

Thread 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.

This button is only enabled when an underlying operating system is supported.

Pane Management controls

Use these buttons to control the panes displayed in the Code window.

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

Processor State group

This field indicates the runtime state of the debug target. This contains:

Unknown

Shows that the target state is not known to the debugger.

Stopped Shows that the target is connected but not active.

Running Shows that an image is currently running. In this case, a running progress indicator is also included.

3.3.2 Editing toolbar

The Editing toolbar, see Figure 3-3 on page 3-9, 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. Hold your mouse pointer over the field to see the full path name.

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.

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.

If the Line number field is empty, click inside this field and press Enter to display the line number where the text insertion point is currently located. Click inside the field again and press Enter to scroll to this location.

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.

Click on the drop-down arrow to access the source control commands (if enabled).

3.3.3 Customizing the Actions toolbar

To customize your Actions toolbar, select **View** → **Toolbars** → **Customize...** from the Code window main menu to display the selection box shown in Figure 3-4.

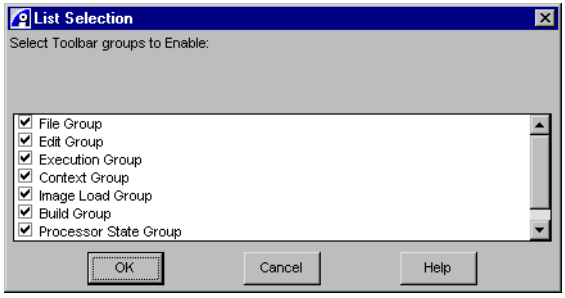


Figure 3-4 Toolbar groups selection box

Check boxes show which toolbar groups are currently enabled on the Actions toolbar. Click a selected check box to disable the associated group. Click **OK** to close the selection box and return to the Code window where the disabled groups have been removed from the button toolbar. To re-enable button groups, display the selection box and click on the check boxes so that they are selected.

If you customize a toolbar, this persists to any new Code windows that you open from this calling window. However, you cannot customize the toolbars shown in a standalone Editor window (see *Editor window* on page 3-20).

Note

Custom toolbar groups are not available in this release.

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*
- *In-place editing* on page 3-17
- *Working with tabs* on page 3-18.

3.4.1 Floating, docking, and resizing windows and panes

Panes are docked to default positions in the Code window when RealView Debugger starts in the default state. You can resize the middle pane row by dragging the upper boundary to the required height. Similarly, drag the left boundary to a new position to enlarge the side pane. You can enlarge the Output pane by dragging the upper boundary.

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

- select **Float** from the **Pane Content** menu
- double-click on the pane title bar.

A floating pane is still tied to the calling window. If, for example, you float a pane from the middle pane row and then click the **Show/Hide Middle Pane** control in the Code window, the floating pane is also hidden. To restore the pane, click the **Show/Hide Middle Pane** control again.

To dock a floating pane, select **Dock** from the **Pane Content** menu or double-click on the pane title bar.

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. You can also move between the File Editor pane and the Output pane using Ctrl+Tab or Shift+Ctrl+Tab.

When working with several Code windows, left-click inside a pane entry to change the focus. 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-pace 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 suspends the current editing operation so that you can complete it when focus returns. Similarly, if you click inside another pane, or change the current view, RealView Debugger cancels the current 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 shows 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.

3.5 Connection Control window

The Connection Control window enables you to manage your debug targets and connections. It includes a tree control, which comprises:

Target execution vehicle

The top-level group is the supported target execution vehicle. It corresponds to the physical connection, and identifies:

- the manufacturer
- the type of physical connection.

For RealView ICE Micro Edition (RVI-ME) it is `ARM-ARM-USB`, which indicates that this vehicle is manufactured by ARM Limited, and is connected to a USB port.

Access-provider connection

The second-level group shows the type of vehicle, or the debug target interface used to support the connection. For RVI-ME, this is the ARM JTAG debug tool for embedded systems.

Endpoint connections

The third-level entry shows the target processors that are available, for example, `ARM7TDMI_0`.

The endpoint connection is accompanied by a check box to show the current state of the connection. When connected, this check box is checked. When disconnected, the check box is blank.

To display the Connection Control window:

1. Start RealView Debugger (see *Starting RealView Debugger* on page 4-6)
2. Select **File** → **Connection** → **Connect to Target...** from the Code window main menu.

For instructions on how to connect to a debug target, see *Connecting to a debug target* on page 4-9.

Also, see the chapter on connecting to targets in the *RealView Developer Kit v1.0 Debugger User Guide*.

3.6 Editor window

RealView Debugger includes a range of editing features to help you work with source code and projects. Use a standalone Editor window to access these editing features so that you can edit files independently of the debugging session or to include these files within your project.

To display a standalone Editor window, select **View → Editor Window** from the Code window main menu. This displays a floating Editor window shown in Figure 3-5.

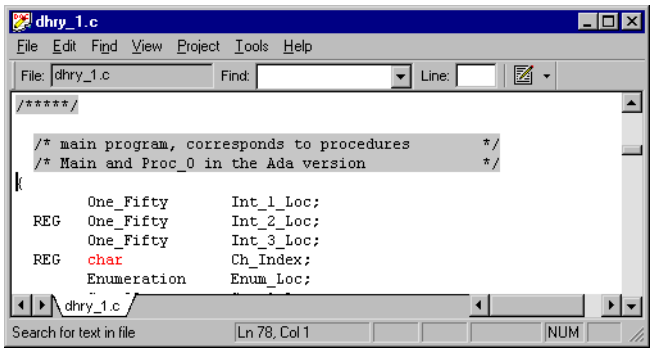


Figure 3-5 Editor Window

In this example, the window is already loaded with the file displayed in the topmost tab in the parent File Editor pane.

A quick way to display an empty Editor window is to click on the **Dsm** tab and then select **View → Editor Window** from the Code window main menu.

Note

You can also display standalone Editor windows from the **Tools** menu.

3.7 Resource Viewer window

The Resource Viewer gives access to all the debugger resources as your debugging session progresses. With the appropriate OS support package, the Resource Viewer window also gives access to RTOS resources including lists of timers, threads, queues, event flags, and memory in byte and block pools. In this mode, additional tabs are displayed.

To display the Resource Viewer window shown in Figure 3-6, start RealView Debugger in the usual way and select **View → Resource Viewer Window** from the Code window main menu.

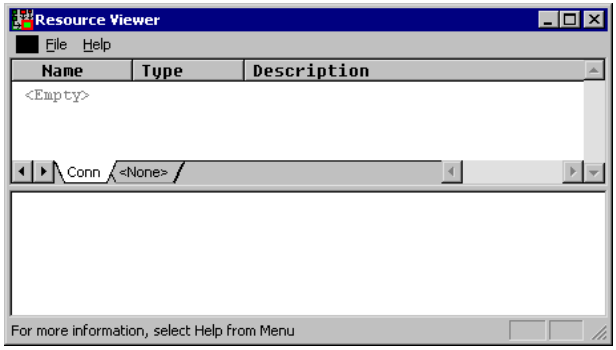


Figure 3-6 Resource Viewer window

The contents of this window change as you establish new target connections, start applications, and debug your images.

The title bar shows the connection, and the Color Box at the left of the window menu bar shows the attachment status of the calling window.

The Resource Viewer window includes:

- *File menu*
- *Help menu* on page 3-22
- *Resources list* on page 3-22
- *Details area* on page 3-23.

3.7.1 File menu

This menu contains:

Update List Rereads the board file and updates the items displayed in the Resources list. This might be necessary if you change your connection without closing the Resource Viewer window.

Display Details

Displays details about a selected entry in the Resources list. A short description is shown in the Details area in the window. You can also display details about an item by double-clicking on the entry in the Resources list.

This is the default display format when you open the Resource Viewer window.

Display Details as Property

Select this option to display details information in a properties box.

Select this option to change the default display format while the window is open. Close the Resource Viewer window to restore the default, that is a description is shown in the Details area.

Clear Log Clears messages and information displayed in the Details area.

Auto Update Details on Stop

Automatically updates the Details area when any image running on the connection stops. This gives you information about the state of the connection when the process terminated. In multiprocessor debugging mode, this applies across all connections.

Auto Update Automatically updates the Resources list as you change debugger resources. Selected by default.

Close Window

Closes the Resource Viewer window.

3.7.2 Help menu

Select **Help** from the Resource Viewer window menu bar to display the **Help** menu.

This menu gives you access to the RealView Debugger online help and displays details of your version of RealView Debugger. You can also use this menu to create and submit a problem report.

3.7.3 Resources list

The Resources list box displays all the resources available to RealView Debugger. In single-processor debugging mode, it contains only the **Conn** tab showing the connection.

If you are debugging a multithreaded application, the Resources list box displays a series of tabs, see the chapter describing RTOS support in *RealView Developer Kit v1.0 Debugger User Guide* for details.

3.7.4 Details area

The bottom half of the Resource Viewer window displays information about a chosen entry.

Chapter 4
Getting Started with RealView Developer Kit

The component products provided with RealView® Developer Kit (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:

- *Building and debugging task overview* on page 4-2
- *Using the example projects* on page 4-4
- *Starting and Exiting RealView Debugger* on page 4-6
- *Opening an existing RealView Debugger project* on page 4-7
- *Connecting to a debug target* on page 4-9
- *Changing your target board/chip definition* on page 4-8
- *Loading an image ready for debugging* on page 4-10
- *Unloading an image* on page 4-12
- *Running the image* on page 4-13
- *Basic debugging tasks with RealView Debugger* on page 4-14
- *Building and rebuilding an image with RealView Debugger* on page 4-17
- *Help on creating RealView Debugger projects* on page 4-18
- *Getting started with the compilation tools* on page 4-23.

4.1 Building and debugging task overview

Table 4-1 is a high-level procedure showing the main tasks for building and debugging applications with the RVDK tools, and where to find the details.

The tasks referred to in the rest of this chapter are not necessarily described in the order presented in Table 4-1. If you are using the RVDK tools for the first time, it is suggested that you work through the tasks in the order described in this chapter. The sequence presented in Table 4-1 reflects the order in which the tasks might usually be performed.

Table 4-1 Main building and debugging tasks

Step	Description	Reference
1	Start RealView Debugger.	*Starting and Exiting RealView Debugger* on page 4-6
2	Configure your debug target and connections as required.	• *Changing your target board/chip definition* on page 4-8 • see the chapters describing connection and target configuration *RealView Developer Kit v1.0 Debugger User Guide.*
3	Connect to your debug target.	*Connecting to a debug target* on page 4-9
4	Decide what image to use: • If you want to load an existing image, such as a prebuilt example image, continue at step 11. • If you want to build a new image, continue at step 5.	*Using the example projects* on page 4-4
5	Decide how to build a new image: • to build using RealView Debugger, continue at step 8 • to build using the RVCT build tools directly, continue at step 6.	
6	If you want to use the RVCT build tools directly, then create makefiles or Windows command files containing the required build commands.	*Getting started with the compilation tools* on page 4-23
7	Build the image, then continue at step 11.	

Table 4-1 Main building and debugging tasks

Step	Description	Reference
8	If a RealView Debugger project already exists, continue at step 9. Otherwise, create a new RealView Debugger project, then continue at step 10.	*Help on creating RealView Debugger projects* on page 4-18
9	Open the existing RealView Debugger project.	*Opening an existing RealView Debugger project* on page 4-7
10	Build the image for the RealView Debugger project.	*Building and rebuilding an image with RealView Debugger* on page 4-17
11	Load an image ready for debugging.	*Loading an image ready for debugging* on page 4-10
12	Prepare any debugging facilities, such as breakpoints.	*Basic debugging tasks with RealView Debugger* on page 4-14
13	Run the image.	*Running the image* on page 4-13
14	Perform the required debugging and monitoring tasks, such as stepping, and displaying contents of variables and memory.	*Basic debugging tasks with RealView Debugger* on page 4-14
15	What is the result of the debugging session? • If there are problems, continue at step 16. • If there are no problems, rebuild your image for final release.	• *Building and rebuilding an image with RealView Debugger* on page 4-17 • *Getting started with the compilation tools* on page 4-23.
16	Decide how to fix any problems in your source code: • use the RealView Debugger editor • use another source editor of your choice.	See the chapter on editing source code in the *RealView Developer Kit v1.0 Debugger User Guide*.
17	When you have fixed the problem, return to step 5 to rebuild, reload, and debug the image.	

4.2 Using the example projects

RVDK provides the following example projects (see *RVDK example projects* on page 1-4):

- Example projects specific to the boards and processors supported by RVDK, which include:
  - source code
  - a RealView Debugger project (`.prj`) file (see *Help on creating RealView Debugger projects* on page 4-18 for more details about RealView Debugger projects)
  - a RealView Debugger workspace (`.aws`) file.
- RealView Debugger examples, which include:
  - source code
  - pre-built images
  - a RealView Debugger project (`.prj`) file (see *Help on creating RealView Debugger projects* on page 4-18 for more details about RealView Debugger projects).
- RealView Compilation Tools (RVCT) examples, which include:
  - source code
  - make files and Windows command files for you to build the images.

  Some projects might require you to build intermediate files, such as object libraries, before you can build the final image.

Although your aim is to build and debug your own application images, 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. However, many tasks described in the user documentation require that you modify the files in the examples. Before you do this, make a backup copy of the example project files and directories.

Because the RVCT examples do not have RealView Debugger project files, you might want to practise creating RealView Debugger projects using the RVCT examples. See *Help on creating RealView Debugger projects* on page 4-18 for more details about RealView Debugger projects.

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

- The RealView Debugger Dhrystone project is in the RealView Debugger 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 4-17.

- The RVCT Dhrystone project is in RVCT examples directory `...\dhry`.

  This contains a makefile and a Windows command file to enable you to build the image. This example project is used in the *Building the RVCT example Dhrystone project* on page 4-30.

4.3 Starting and Exiting RealView Debugger

This section includes:

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

4.3.1 Starting RealView Debugger

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

Programs → ARM → ARM RealView Developer Kit for ST v1.0 → RealView Debugger v1.6.1

The RealView Debugger splash screen is displayed (see *Splash screen* on page 3-2), and is then replaced by the RealView Debugger Code window.

The first time you start RealView Debugger after installation, the RealView Debugger Code window appears as shown in Figure 3-1 on page 3-3.

4.3.2 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.

4.4 Opening an existing RealView Debugger project

To open an existing RealView Debugger project file:

1. Start RealView Debugger (see *Starting RealView Debugger* on page 4-6).
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 4-21 for details)
- perform automatic operations when a project is opened (see *Automatic operations performed by a project* on page 4-21 for details).

For more details on opening projects, see the chapter on managing projects in the *RealView Developer Kit v1.0 Debugger User Guide*.

4.5 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 v1.0 Debugger User Guide*.

4.5.1 How to change the board/chip definition

To configure the debug target:

1. Display the Connection Control window.

   Select the menu option **File → Connection... → 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 **File → Connection → 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.

4.6 Connecting to a debug target

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

- *Making a connection*
- *Configuring your debug target*
- *Setting connect mode*.

4.6.1 Making a connection

To connect to a debug target, do the following:

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

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

   The Connection Control window is displayed (see *Connection Control window* on page 3-19 for details). If there are currently no connections to the debug target, the target execution vehicle (`ARM-ARM-USB`) is expanded to show the available access-provider connections.

   For more details on connecting to a debug target, see the chapter on connecting to targets in the *RealView Developer Kit v1.0 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 the debug target is established, and the Code window is updated with details for the connection.

4.6.2 Configuring your debug target

RVDK comes with a preconfigured debug target. However, you can configure the RVI-ME debug target to your specific requirements. For more details, see *How to change the board/chip definition* on page 4-8. Also, see the chapters on configuring targets and connections in the *RealView Developer Kit v1.0 Debugger User Guide*.

4.6.3 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 v1.0 Debugger User Guide*.

4.7 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.

4.7.1 Loading an image directly

To load an image directly to your debug target:

1. Start RealView Debugger (see *Starting RealView Debugger* on page 4-6).
2. Connect to the debug target (see *Connecting to a debug target* on page 4-9).
3. Select the menu option **File → 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 v1.0 Debugger User Guide* for more details on auto-projects.

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

4.7.2 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 4-6).
2. Connect to the debug target (see *Connecting to a debug target* on page 4-9).
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 4-7 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 v1.0 Debugger User Guide*.

4.7.3 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 v1.0 Debugger User Guide*.

4.7.4 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 v1.0 Debugger User Guide*.

4.7.5 Reloading an image

During your debugging session you might have to amend your source code and then recompile. Select **File → 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.

4.8 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).

4.8.1 How to explicitly unload an image

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

1. Select **View → Pane Views → Process Control Pane** 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 Image+Symbols` entry, 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**.

4.9 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** → **ExecutionControl** → **Go (Start Execution)** from the main menu.
   -  Click the **Go** button on the Actions toolbar.

4.10 Basic debugging tasks with RealView Debugger

The basic debugging tasks you can perform with RealView Debugger are shown in Table 4-2. The references give detailed instructions for these tasks.

Table 4-2 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 4-16, and the chapter on working with browsers in the *RealView Developer Kit v1.0 Debugger User Guide*
Examining different code views	the chapter on controlling execution in the *RealView Developer Kit v1.0 Debugger User Guide*
Viewing target status	*Actions toolbar* on page 3-13 for a description of the Processor State group
Displaying register contents	the chapter on monitoring execution in the *RealView Developer Kit v1.0 Debugger User Guide*
Changing register contents	the chapter on reading and writing memory, registers, and flash in the *RealView Developer Kit v1.0 Debugger User Guide*
Using the Process Control pane	the chapter on working with images in the *RealView Developer Kit v1.0 Debugger User Guide*
Displaying memory contents	the chapter on monitoring execution in the *RealView Developer Kit v1.0 Debugger User Guide*
Using Tooltip Evaluation to examine variables and register contents	the chapter on editing source code in the *RealView Developer Kit v1.0 Debugger User Guide*
Using the call stack	the chapter on monitoring execution in the *RealView Developer Kit v1.0 Debugger User Guide*
Using browsers and lists	the chapter on working with browsers in the *RealView Developer Kit v1.0 Debugger User Guide*
Setting watches	the chapter on monitoring execution in the *RealView Developer Kit v1.0 Debugger User Guide*

4.10.1 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

4.10.2 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 v1.0 Debugger User Guide* for more details.

4.10.3 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 a debug target* on page 4-9).
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.

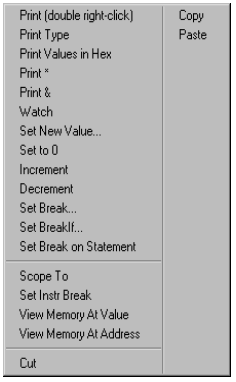


Figure 4-1 Source Variable Name menu

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.

4.11 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 4-23.

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 4-6).
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 4-7 for instructions on how to open a project.
4. To rebuild the `dhrystone.axf` image, select the menu option **Tools → Rebuild All (Clean+Build)**.

   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 4-19 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 v1.0 Debugger User Guide* for more details on building your application.

4.12 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 4-19
- *Limitations of Standard and Library projects* on page 4-20
- *Making a project and a connection interdependent* on page 4-21
- *Automatic operations performed by a project* on page 4-21.

For instructions on how to create a project, see the chapter on managing projects in the *RealView Developer Kit v1.0 Debugger User Guide*.

4.12.1 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 4-20).

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 4-20).

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 4-20).

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.

4.12.2 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 v1.0 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 4-23 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 v1.0 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.

4.12.3 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.

4.12.4 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 v1.0 Debugger User Guide*.
Project	`Specific_device`	This is a setting that you specify for a project. It identifies a specific processor (for example, ARM7TDMI) 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 v1.0 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*).

4.12.5 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 4-21), 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 4-21), 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 v1.0 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 4-19). 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 v1.0 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 v1.0 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 v1.0 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 v1.0 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 v1.0 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 v1.0 Debugger User Guide*

4.13 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 4-24
- *Targeting the ARM or Thumb instruction set* on page 4-25
- *Targeting a specific ARM architecture or processor* on page 4-26
- *Targeting specific procedure call standard variants* on page 4-26
- *Generating debug information* on page 4-27
- *Optimizing your compiled sources* on page 4-27
- *Building an image with a single compiler invocation* on page 4-28
- *Building an image with separate command invocations* on page 4-28
- *Specifying the initial entry point for an image* on page 4-28
- *Creating object libraries* on page 4-29
- *Converting images to binary files* on page 4-29
- *Creating an image memory map with scatter-loading* on page 4-29
- *Building the RVCT example Dhrystone project* on page 4-30.

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

- *RealView Developer Kit v1.0 Assembler Guide*
- *RealView Developer Kit v1.0 Compiler and Libraries Guide*
- *RealView Developer Kit v1.0 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 v1.0 Debugger User Guide*. Also, see *Help on creating RealView Debugger projects* on page 4-18, which provides information to help you decide the type of RealView Debugger project to create.

4.13.1 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.

4.13.2 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`	global, applied to all specified sources
ARM	`#pragma arm`	`CODE32` directive	per-function[a]
Thumb	`armcc --thumb`	`armasm -16`	global, applied to all specified sources
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 4-26 for details.

To see how the assembler directives are used, examine the RVCT 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`

4.13.3 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 4T` Targets ARM architecture v4T.

`--cpu ARM7TDMI`

Targets the ARM7TDMI® processor.

Examine the RVCT example projects *examples_directory*`\cached_dhry\9`*xx*`dhry` to see how this option is used.

For more details about the `--cpu` option, see the *RealView Developer Kit v1.0 Assembler Guide* and the *RealView Developer Kit v1.0 Compiler and Libraries Guide*.

4.13.4 Targeting specific procedure call standard variants

You can specify one or more procedure call standard variants.

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 RVCT example project *examples_directory*`\interwork` to see how this option is used.

Read-only position independence

To specify read-only position independence when compiling or assembling, use the option:

```
--apcs /ropi
```

You can specify read-only position independence when linking with the `-ropi` and `-ro-base` options (see the *RealView Developer Kit v1.0 Linker and Utilities Guide* for more details).

Examine the RVCT example project *examples_directory*`\picpid` to see how these options are used.

Read/write position independence

To specify read/write position independence when compiling or assembling, use the option:

```
--apcs /rwpi
```


You can specify read/write position independence when linking with the `-rwpi` and `-rw-base` options (see the *RealView Developer Kit v1.0 Linker and Utilities Guide* for more details).

Examine the RVCT example project *`examples_directory`*`\picpid` to see how these options are used.

For more details about these, and other `--apcs` options, see the *RealView Developer Kit v1.0 Assembler Guide* and the *RealView Developer Kit v1.0 Compiler and Libraries Guide*.

4.13.5 Generating debug information

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

The compiler also provides a pragma equivalent, `#pragma debug`.

By default, the linker includes debug information (see the description of the `-debug` option in the *RealView Developer Kit v1.0 Linker and Utilities Guide*).

4.13.6 Optimizing your compiled sources

When you are compiling C or C++ code, the compiler enables you to optimize the generated code. Some optimizations are performed by default. However, you can control which optimizations you want to apply to your code.

The compiler also provides pragma equivalents for these optimizations.

For more details on the optimizations that are available, see the *RealView Developer Kit v1.0 Compiler and Libraries Guide*.

4.13.7 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 v1.0 Compiler and Libraries Guide*.

4.13.8 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 generate object files only. For example:

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

   For more details, see the *RealView Developer Kit v1.0 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 v1.0 Linker and Utilities Guide*.

4.13.9 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. To do this, you must perform a separate link step, and use the `-entry` linker option. For example:

```
armlink -entry 0x8000
```

Examine the RVCT example projects *`examples_directory`*`\cached_dhry\9xxdhry` to see how this option is used.

For more details, see the *RealView Developer Kit v1.0 Linker and Utilities Guide*.

4.13.10 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 v1.0 Linker and Utilities Guide*.

4.13.11 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
```

Examine the RVCT example project *examples_directory*`\picpid` to see how the `fromelf` utility is used.

For more information on using the fromELF utility, see the *RealView Developer Kit v1.0 Linker and Utilities Guide*.

4.13.12 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 RVCT 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 v1.0 Linker and Utilities Guide*.

4.13.13 Building the RVCT example Dhrystone project

You can build the RVCT 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
```

Note

Because both source files are written in C, the compiler automatically compiles using ISO standard C. Therefore, the `--c90` compiler option is omitted.

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 v1.0 Compiler and Libraries Guide* for a description of the compiler options.

See the *RealView Developer Kit v1.0 Linker and Utilities Guide* for a description of the linker options.

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.

ATPE
: *See* ARM Toolkit Proprietary ELF.

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 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.

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.

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.

Debugger An application that monitors and controls the execution of a second application. Usually used to find errors in the application program flow.

DWARF 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.

Read-Only Position Independent (ROPI)
: Code and read-only data addresses can be changed at run-time.

Read/Write Position Independent (RWPI)
: Read/write data addresses can be changed at run-time.

RealView ICE Micro Edition
: A multi-processor JTAG-based debug tool for embedded systems.

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.

ROPI
: *See* Read Only Position Independent.

RTOS
: Real-Time Operating System.

RWPI
: *See* Read Write Position Independent.

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

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

Access-provider connection 3-19
ANSI C library 2-4
ar 2-4
ARM Toolit Proprietary ELF Format 2-2
armar 2-4
ARM-ARM-USB vehicle 3-19
armasm 2-3
armcc 2-3
armlink 2-3
ATPE
  see ARM Toolit Proprietary ELF Format
Autobinding 4-21
Automatic operations
  setting up 4-22
  summary 4-21

B

Binding 4-21
  autobinding 4-21
  default binding 4-22
  specific device binding 4-21
Books
  Linker and Utilities Guide vii
Breakpoints
  setting simple 4-15
Build target configurations 4-19

C

Cancel command 3-13
Code views 3-5
Code window 3-16
  Actions toolbar 3-8, 3-13
  attachment 3-4
  Button toolbars 3-13
  button toolbars 3-8
  CAPS 3-10
  changing panes 3-7
Color Box 3-9
components 3-3
Cursor location field 3-10
customizing toolbars 3-15
debug views 3-5
default 3-2, 3-3
default panes 3-5
docking panes 3-16
Editing toolbar 3-8, 3-13, 3-14
Editor Window 3-20
File Editor pane 3-5
floating panes 3-16
focus control 3-16
in-place editing 3-17
JOU 3-10
LOG 3-10
Main menu 3-11
NUM 3-10
pane controls 3-7
panes 3-5
Resource Viewer 3-21
Status line 3-9

STDIOlog 3-10
title bar 3-4
windows and panes 3-5
Color Box 3-9
Command line
development tools 2-3
Components 2-3
Concepts
debug target 2-6
multithreading 2-7
RTOS 2-7
Concepts and terminology 2-6
see also Glossary
Connection Control window
access-provider connection 3-19
endpoint connections 3-19
target execution vehicle 3-19
Connections
interdependence with projects 4-21
Container projects 4-18
Custom projects 4-18
Customizing
toolbars 3-15

D

Debug target
concept 2-6
Default binding 4-22
Desktop
Actions toolbar 3-8, 3-13
attaching windows 3-4
bottom pane 3-7
Call Stack pane 3-6
CAPS 3-10
changing panes 3-7
Code window 3-2, 3-3
Collapse Pane 3-8
Color Box 3-9
Cursor location field 3-10
customizing toolbars 3-15
docking panes 3-16
Edit menu 3-11
Editing toolbar 3-8, 3-14
Expand Pane 3-8
File Editor pane 3-5
File menu 3-11
Find menu 3-12
floating panes 3-16
floating toolbars 3-9
focus 3-16
formatting views 3-7
hiding toolbars 3-9
in-place editing 3-17
JOU 3-10
LOG 3-10
Main menu 3-11
Memory pane 3-6
middle row pane 3-6
NUM 3-10
Pane Content menu 3-8
pane controls 3-7
Pane Menu 3-8
Processor State 3-14
Project menu 3-12
Register pane 3-6
side pane 3-6
splash screen 3-2
Status line 3-9
STDIOlog 3-10
tabs 3-18
View menu 3-12
viewing toolbars 3-8
Watch pane 3-6
Documentation
PDF 1-6
Downloads
RTOS Awareness 2-8
DWARF2 2-4

E

EABI 2-5
Editor Window 3-20
Embedded Application Binary Interface 2-5
Endpoint connections 3-19

F

File Editor
File Editor pane 3-5
Focus
in Code window 3-16
fromELF 2-4

I

Images
loading directly 4-10
loading from a project 4-10
loading multi-image applications 4-11
reloading 4-11
unloading 4-12
In-place editing 3-17
focus 3-17

L

Libraries
support 2-4
Library projects 4-18

M

Menus
Edit 3-11
File 3-11
Find 3-12
Main 3-11
Pane Content 3-8
Pane Menu 3-8
Pane Views 3-7
Project 3-12
View 3-12
Multithreading
concept 2-7

P

Panes
bottom 3-7
Call Stack 3-6
changing 3-7
collapsing 3-8
docking 3-16
expanding 3-8
File Editor 3-5
floating 3-16
focus 3-16
Memory 3-6

middle row 3-6
Register 3-6
side 3-6
Watch 3-6
Projects
autobinding 4-21
automatic operations 4-21
binding 4-21
bound 3-4
build target configurations 4-19
Container 4-18
Custom 4-18
default binding 4-22
interdependence with connections 4-21
Library 4-18
limitations of 4-20
opening 4-7
specific device binding 4-21
Standard 4-18
unbound 3-4

R

Real Time Operating Systems 2-7
RealView Debugger
Code window 3-2, 3-3
concepts 2-6
desktop 3-2
image reloading 4-11
image unloading 4-12
window attachment 3-4
Reloading
images 4-11
Resource Viewer 3-21
Rogue Wave C++ library 2-4
RTOS 2-7
Awareness Downloads,goto 2-8
concept 2-7
support package 2-8
threads 2-7
viewing resources 3-21

S

Software Problem Report 3-12
Specific device binding 4-21
SPR
see Software Problem Report
Standard projects 4-18
Standards 2-4
Support for RTOS 2-8

T

Tabbed pages 3-18
Tabs 3-18
Target execution vehicle 3-19
Terminology and concepts 2-6
Threads 2-7
Toolbars
Actions 3-13
Build group 3-14
Command cancel 3-13
Context controls 3-13
customizing 3-15
Edit group 3-13
Editing 3-14
Execution group 3-13
File field 3-14
File group 3-13
Find field 3-14
Image load group 3-13
Line number field 3-14
Pane Management controls 3-14
Processor State group 3-14
Source control button 3-15
Thread button 3-14

U

Unloading
images 4-12

V

Vehicle
ARM-ARM-USB 3-19

W

Windows
attachment 3-4
Connection Control 3-19
docking 3-16
Editor 3-20
floating 3-16
focus 3-16
Resource Viewer 3-21