RealView® Compilation Tools

Version 2.1

Essentials Guide

Printed on: February 2, 2004
RealView Compilation Tools
Essentials Guide

Copyright © 2002-2004 ARM Limited. All rights reserved.

Release Information
The following changes have been made to this book.

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>August 2002</td>
<td>A</td>
<td>Release 1.2</td>
</tr>
<tr>
<td>January 2003</td>
<td>B</td>
<td>Release 2.0</td>
</tr>
<tr>
<td>September 2003</td>
<td>C</td>
<td>Release 2.0.1 for RVDS v2.0</td>
</tr>
<tr>
<td>January 2004</td>
<td>D</td>
<td>Release 2.1 for RVDS v2.1</td>
</tr>
</tbody>
</table>

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Confidentiality Status

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

Product Status

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

Web Address

http://www.arm.com
Contents
RealView Compilation Tools Essentials Guide

Preface
About this book ........................................................................................................ vi
Feedback .................................................................................................................. ix

Chapter 1 Introduction
1.1 About RealView Compilation Tools ................................................................. 1-2
1.2 Getting more information online .................................................................... 1-5

Chapter 2 Differences
2.1 Overview ......................................................................................................... 2-2
2.2 Changes between RVCT v2.1 and RVCT v2.0 .............................................. 2-3
2.3 Changes between RVCT v2.0 and RVCT v1.2 .............................................. 2-7

Chapter 3 Creating an Application
3.1 Building an application ................................................................................... 3-2
3.2 Using ARM libraries ....................................................................................... 3-8
3.3 Using your own libraries ................................................................................ 3-11

Glossary
Preface

This preface introduces the RealView® Compilation Tools v2.1 Essentials Guide and other user documentation. It contains the following sections:

- About this book on page vi
- Feedback on page ix.
About this book

This book provides an overview of the RealView Compilation Tools (RVCT) v2.1 tools and documentation.

Intended audience

This book is written for all developers who are producing applications using RVCT. It assumes that you are an experienced software developer.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction

Read this chapter for an introduction to RVCT. The components of RVCT and the online documentation are described.

Chapter 2 Differences

Read this chapter for details of the differences between RVCT v2.1, RVCT v2.0, and RVCT v1.2.

Chapter 3 Creating an Application

Read this chapter for a brief overview of how to create an application using RVCT.

Glossary

An alphabetically arranged glossary defines the special terms used in this book.

This book assumes that you have installed your ARM® software in the default location for example, on Windows this might be volume:\Program Files\ARM. This is assumed to be the location of install_directory when referring to path names, for example install_directory\Documentation\Specifications\.... You might have to change this if you have installed your ARM software in a different location.

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.
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 and addenda, and the ARM Frequently Asked Questions.

ARM publications

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

- **RealView Compilation Tools v2.1 Developer Guide** (ARM DUI 0203). This book provides tutorial information on writing code targeted at the ARM family of processors.


- **RealView Compilation Tools v2.1 Compiler and Libraries Guide** (ARM DUI 0205). This book provides reference information for RVCT. It describes the command-line options to the compiler and gives reference material on the ARM implementation of the C and C++ compiler and the C libraries.

- **RealView Compilation Tools v2.1 Linker and Utilities Guide** (ARM DUI 0206). This book provides reference information on the command-line options to the ARM linker and the fromELF utility.

For general information on software interfaces and standards supported by ARM, see install_directory\Documentation\Specifications\...
In addition, refer to the following documentation for specific information relating to ARM products:

- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- the ARM datasheet or technical reference manual for your hardware device.

**Other publications**

For a comprehensive introduction to ARM architecture see:

Feedback

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

Feedback on the product

If you have any problems with this product, 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.
Preface
Chapter 1

Introduction

This chapter introduces RealView® Compilation Tools v2.1 (RVCT v2.1) and describes its software components and documentation. It contains the following sections:

- About RealView Compilation Tools on page 1-2
- Getting more information online on page 1-5.
1.1 About RealView Compilation Tools

RVCT consists of a suite of tools, together with supporting documentation and examples, that enable you to write and build applications for the ARM® family of RISC processors.

You can use RVCT to build C, C++, or ARM assembly language programs.

1.1.1 Components of RVCT

RVCT consists of the following major components:

- **Development tools**
- **Supported standards** on page 1-3
- **Supporting software** on page 1-4.

**Development tools**

The following development tools are provided:

- **armcc** The ARM and Thumb® C and C++ compiler. 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.

- **armsm** The ARM and Thumb assembler. This assembles both ARM assembly language and Thumb assembly language source.

- **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 ELF 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 HTML documentation on the CD ROM.

**C++ runtime libraries**

The ARM C++ runtime libraries enable support for core C++ language features, as provided in the C++ header files `<new`, `<typeinfo`, and `<exception` in `install_directory\RVCT\Data\include`. **
Introduction

C library The ARM C library provides an implementation of the library features as defined in the ISO/IEC 9899:1990, C Standard and the Normative Addendum 1 from 1994 (with the exception of file-IO).

fromELF The ARM image conversion utility. This accepts ELF format 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 also generate text information about the input image, such as disassembly and its code and data size.

armar The ARM librarian enables sets of ELF format object files to be collected together and maintained in libraries. You can pass such a library to the linker in place of several ELF 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 all the tools in the RVCT suite, and by ELF/DWARF2 compatible debuggers from ARM, for example ARM eXtended Debugger (AXD) and RealView Debugger.

ISO C The ARM compiler accepts ISO/IEC 9899:1990 C, including the Normative Addendum 1 from 1994 (except file-IO), as input. The option --strict can be used to enforce strict ISO compliance.

C++ The ARM compiler supports the full ISO/IEC 14822:1998 C++ language, with the exception of the export keyword.

ELF The ARM tools produce files in ELF format. The fromELF utility can translate ELF files into other formats.

BSABI By conforming to the Application Binary Interface (ABI) for the ARM Architecture (base standard) [BSABI], ARM and Thumb objects and libraries from different producers can work together. For more details, see http://www.arm.com.
Supporting software

To debug your programs under simulation, or on hardware that is based on an ARM core, use an ELF/DWARF2 compatible debugger, for example ARM RealView Debugger or AXD.

To debug your programs under simulation, use the RealView ARMulator® ISS supporting software.

RealView ARMulator ISS

RealView ARMulator ISS (RVISS) is an Instruction Set Simulator (ISS) that is supplied with RealView Developer Suite. It communicates with a debugger using either Remote Debug Interface (RDI) or RV-msg using the RealView Connection Broker. It can run on the same host computer as the debugger, or on a system remote from that running the debugger.

RVISS is a collection of programs that simulate the instruction sets and architecture of various ARM processors. This provides instruction-accurate simulation and enables ARM and Thumb executable programs to be run on non-native hardware.

RVISS provides modules that model:

- the ARM processor core
- the memory used by the processor.

There are alternative predefined models for each of these parts. However, you can create your own models if a supplied model does not meet your requirements.

For more details, see RealView ARMulator ISS User Guide.
1.2 Getting more information online

Depending on your installation, the full documentation suite is available online as PDF files or DynaText. What is installed depends on your working environment, for example, you can choose to install PDF or DynaText on Windows, or PDF on Sun Solaris.

The DynaText browser is installed by default for a Typical installation.

The PDF and DynaText files contain the same information.

In addition, documentation for the Rogue Wave C++ library is available in HTML format on all supported platforms. This is installed by default for a Typical installation.

For more details, see:
- Using RVCT on Windows
- Using RVCT on Sun Solaris and Red Hat Linux
- Rogue Wave documentation on page 1-6.

1.2.1 Using RVCT on Windows

Select Programs → ARM from the Windows Start menu. From here select either:
- DynaText Documentation to view the DynaText files
- RealView Developer Suite v2.1 → PDF Documentation to view the PDF files.

Unless you change the defaults, the PDF and DynaText files are installed in 

\( \text{install\_directory/Documentation/RVCT/2.1/release/windows} \)

1.2.2 Using RVCT on Sun Solaris and Red Hat Linux

The full documentation suite is available as PDF and DynaText files for developers working on Sun Solaris and on Red Hat Linux. Where you have set up desktop links, use these to access the required documentation.

Unless you change the defaults, the PDF and DynaText files are installed in 

\( \text{install\_directory/Documentation/RVCT/2.1/release/unix} \)

--- Note ---

DynaText files are not installed for developers working on Red Hat Linux.
1.2.3 Rogue Wave documentation

The manuals for the Rogue Wave Standard C++ Library for RVCT are provided on the product CD ROM as HTML files. Use a web browser, such as Netscape or Internet Explorer, to view these files. For example, select the file install_directory\Documentation\RogueWave\1.0\release\stdref\index.htm to display the HTML documentation for Rogue Wave (see Figure 1-1 where the install_directory is D:\ARM).

![HTML browser](image)

**Standard C++ Library Class Reference**

Welcome to the Standard C++ Library Class Reference. Click on the "Chapter 2: Reference" link below to see a hypertext list of class descriptions.

For member functions and data types (exclusive of constructors and destructors), you can consult the comprehensive index. Each class description has an index specific to the class.

- Chapter 1: Introduction
- Chapter 2: Reference

![Figure 1-1 HTML browser](image)
Chapter 2
Differences

This chapter describes the major differences between RealView® Compilation Tools (RVCT) v2.1, RVCT v2.0, and RVCT v1.2. It contains the following sections:

- Overview on page 2-2
- Changes between RVCT v2.1 and RVCT v2.0 on page 2-3
- Changes between RVCT v2.0 and RVCT v1.2 on page 2-7.
2.1 Overview

The most important differences between RVCT v2.1 and RVCT v2.0 are:

- C++ exception handling is fully supported by the compiler.
- Support for new cores, such as ARM1136.
- Unused virtual function elimination for C++ code.
- Multifile compilation enabling global optimizations across source files.
- Linker feedback to the compiler regarding unused functions.
- Linker support to generate exception tables for legacy objects.
- Read/Write data compression/decompression.
- Support for some GCC source language extensions.
- The assembler examines instructions to decide whether code should be marked as PRE58. Where required, this change is made automatically.
- Extended compliance with the ABI for the ARM Architecture (base standard) [BSABI]. For more information, see http://www.arm.com.

For full details on the changes between previous releases see:

- Deprecated features.
- Changes between RVCT v2.1 and RVCT v2.0 on page 2-3.

2.1.1 Deprecated features

Be aware of the following changes in RVCT v2.1:

- The ARM linker accepts object files in ARM Object Format (AOF) format and libraries in ARM Library Format (ALF) format. However, these legacy Software Development Toolkit (SDT) formats are obsolete and will not be supported in the future.
- The use of single dashes for keywords, for example arm\link -help, is deprecated and will not be supported in the future. Use double dashes when working with the compilation tools, for example arm\link --help.
- The compiler options --fpu fpa and --fpu softfpa are deprecated and will not be supported in the future.
- Support for VFPv1 is deprecated.
2.2 Changes between RVCT v2.1 and RVCT v2.0

This section describes the changes between RVCT v2.1 and RVCT v2.0, and includes:

- General changes
- Changes to the ARM compiler on page 2-4
- Changes to library support on page 2-4
- Changes to the ARM linker on page 2-5
- Changes to the ARM assembler on page 2-6
- Changes to the fromELF utility on page 2-6.

2.2.1 General changes

The following changes have been made to RVCT v2.1:

- Enhanced support for ARMv6 cores. To see a full list of supported cores use:
  
  armcc --cpu list

- The compiler, linker, assembler, and fromelf utility support the new --diag_style option to generate warnings and errors in a format that is more compatible with IDEs, such as Microsoft Visual Studio.

- The compiler and linker support the new facility to remove unused virtual functions from generated C++ code.

- The ARM tools can share suitable strings across compile units using SHF_STRINGS sections, as defined by the ABI for the ARM Architecture, ELF standard (AAELF).

- Support for VFPv1 is deprecated. The new default is VFPv2. To see a full list of supported FPUs use:

  armcc --fpu list

- The use of single dashes for keywords is deprecated and will not be supported in the future. Use double dashes when working with the compilation tools.

- By default, the compilation tools warn against the use of deprecated options (such as the compiler option --fpu softfp). You can change this behavior by setting the environment variable RVCT21_CLWARN to one of the following values:

  0 Warn against old syntax and deprecated options.
  1 Accept old syntax without a warning, but warn against deprecated options. This is the default.
  2 Accept old syntax and deprecated options without a warning.
2.2.2 Changes to the ARM compiler

The following changes have been made to RVCT v2.1:

- Support for GNU extensions when you run the compiler with the --gnu option. However, some extensions are also supported when you run the compiler without this option. These compilation modes are referred to as:

  **ARM mode**
  
  The default mode, that is compilations without the --gnu option.

  **GNU mode**
  
  Compilations with the --gnu option.

For a complete list of all GNU extensions, and the mode and language in which they are supported, see the chapter describing the compiler reference in *RealView Compilation Tools v2.1 Compiler and Libraries Guide*.

- Multifile compilation provides optimization across compile units. Use the new --multifile option to specify this behavior. Multifile compilation requires you to specify multiple files on the command line, for example:

  armcc [options] --multifile ifile_1 ... ifile_n

- New -O3 optimization level, which includes Multifile compilation by default.

- Noreturn functions.

- Enhanced support for ISO C++ through the Edison Design Group (EDG) front end. This provides a full C++ parser that passes a program representation to the ARM compiler for code generation. This now includes support for throwing and catching C++ exceptions.

- New --cpu list and --fpu list options to display details about supported CPUs and architectures.

- New __breakpoint() intrinsic.

2.2.3 Changes to library support

The following changes have been made to RVCT v2.1:

- The C++ libraries, that is the Rogue Wave and C++ runtime libraries, now support C++ exceptions. The C++ libraries continue to support applications that do not require exceptions support.

- The C library now supports all of the schar.h function, except file-IO, and the c99 hexadecimal floating point support in printf and scanf.
A new region tables format has been introduced to support compression algorithms. This new format no longer contains ZISection$$Table.

### 2.2.4 Changes to the ARM linker

The following changes have been made to RVCT v2.1:

- Linker feedback is available, for the next time a file is compiled, to inform the compiler about unused functions. These are placed in their own sections for future elimination by the linker.

- Read/Write data compression is enabled by default to optimize ROM size.

- A new option is available, --rosplit, to output two RO execution regions, one for code and one for data.

- New command-line options are available to support C++ exception tables. Use the new option --noexceptions to ensure that your code is exceptions free.

  The new option --exceptions_tables=unwind|nounwind forces the linker to generate exception tables regardless of the content of the input files. The linker can create exception tables for C and assembly language objects with debug frame information using, for example, --exceptions_tables=unwind.

- A new option is available, --userlibpath, to specify where to search for user libraries.

- The linker is now stricter in checking alignment in object files. It ensures that any code that requires eight-byte alignment of the stack is only called, directly or indirectly, by code that preserves eight-byte alignment of the stack. The linker generates an error message if a stack alignment conflict is detected:

  Error L6238E: object_name.o(section_name) contains invalid call from '-PRES8' function to 'REQ8' function_name

  A similar warning message is generated where the address of an external symbol is referenced:

  Warning L6306W: '-PRES8' section object_name.o(section_name) should not use the address of 'REQ8' function_name

  To correct this problem:

  — Rebuild all your object files and libraries using RVCT v2.1, if possible. If you have assembly language files, check that they preserve eight-byte alignment and correct them if required.

  — Where you cannot rebuild legacy code, use fromelf -c to disassemble the object code and then check that it preserves eight-byte alignment.
For full details on this change see the section on stack alignment in the chapter introducing the linker in *RealView Compilation Tools v2.1 Linker and Utilities Guide*.

### 2.2.5 Changes to the ARM assembler

The following changes have been made to RVCT v2.1:

- New `--cpu list` and `--fpu list` options to display details about CPU and architectures supported.

- The assembler examines instructions that modify the Stack Pointer (SP) to decide whether code should be marked as _PRES8_. Where required, this change is made automatically (see the chapter describing the directives reference in *RealView Compilation Tools v2.1 Assembler Guide*).

### 2.2.6 Changes to the fromELF utility

The following change has been made to RVCT v2.1:

- New `--expandarrays` option to decode an ELF image so that arrays are expanded both inside and outside structures. This option can only be used in conjunction with the `--text=/a` option.
2.3 Changes between RVCT v2.0 and RVCT v1.2

This section describes the changes between RVCT v2.0 and RVCT v1.2, and includes:

- General changes
- Changes to the ARM compiler
- Changes to the ARM linker on page 2-8
- Changes to the ARM assembler on page 2-9.

2.3.1 General changes

The following changes have been made to RVCT v2.0:

- Support for ARM Architecture v6.
- Compliance with the ABI for the ARM Architecture (base standard) [BSABI]. For more information, see http://www.arm.com.
- For floating point exceptions to occur, you must select --fmode ieee_full. This is because the default setting is now --fmode std and so floating point exceptions are not generated by default.
- Support for double dashes "--" to indicate command-line keywords (for example, --cpp).

2.3.2 Changes to the ARM compiler

The major changes that have been made to the ARM compiler (armcc) are as follows:

- There is a new front-end to the RVCT v2.0 compiler that includes changes to the command-line options. The options available in the older ARM compilers are supported for backwards compatibility.
- The four individual compilers, armcc, tcc, armcpp and tcpp, are now merged into a single compiler, armcc. However, to aid migration to the new compiler, you can invoke the RVCT v2.0 compiler using the individual compiler names.
- Support for ARMv6, and exploits the unaligned access behavior of ARMv6.
- A new embedded assembler to complement the inline assembler.
- ARM and Thumb compilation on a per-function basis, using #pragma arm and #pragma thumb.
- Five floating-point models using the --fmode option.
- The behavior of the --list option is different from that in the older compilers.
Differences

- C++ template instantiation.
- C++ namespaces.
- You can specify the level of pointer alignment.
- Control and manipulation of diagnostic messages. Also, the numbering of diagnostic messages has changed. Messages now have the number format #nnnn or #nnn-D. The message numbers for messages with the -D suffix can be used in those options that enable you to manipulate the diagnostic messages.
- Many old compiler options are not supported in the new interface. However, for backwards compatibility, these options are available if you use the --old_cfe option. See the appendix describing the older compiler options in RealView Compilation Tools v2.1 Compiler and Libraries Guide for more details. Where applicable, this appendix also shows how the old compiler options map to the new compiler options. For those messages listed in RealView Compilation Tools v2.1 Compiler and Libraries Guide, the appendix also shows the equivalent messages that are output by the new compiler interface.

Note

If you use the --old_cfe option, then the older numbering format is used for messages output by the compiler.

Other changes include the addition of new pragmas and predefined macros, additional C and C++ language extensions, and changes to the ARM C and C++ libraries.

2.3.3 Changes to the ARM linker

The following changes have been made to the ARM linker (armlink):

- The --unresolved option is now applicable to partial linking.
- A new steering file command, RESOLVE, has been added, and is used when performing partial linking. RESOLVE is similar in use to the armlink option --unresolved.
- The option --edit now accepts multiple files.
- There is a new option --pad to specify a value for padding bytes.
- New scatter-loading attributes, EMPTY and ZEROPAD, have been added.
2.3.4 Changes to the ARM assembler

The following changes have been made to the ARM assembler (armasm):

- Support for new ARM architecture v6 instructions has been added. These include saturating instructions, parallel instructions, and packing and unpacking instructions.

- The ALIGN directive has an additional parameter, to specify the contents of any padding. This parameter is optional.

- There is a new AREA directive, NOALLOC.

- There are two new directives, ELIF and FRAME RETURN ADDRESS.

- There are four new built-in variables \{AREANAME\}, \{COMMANDLINE\}, \{LINENUM\}, and \{INPUTFILE\}. 
Chapter 3
Creating an Application

This chapter describes how to create an application using RealView® Compilation Tools v2.1 (RVCT v2.1). It contains the following sections:

- Building an application on page 3-2
- Using ARM libraries on page 3-8
- Using your own libraries on page 3-11.
3.1 Building an application

This section describes how to build an application. See:

- Using the ARM compilers
- Using the ARM assembler on page 3-6.

3.1.1 Using the ARM compilers

The ARM® compiler, armcc, can compile C and C++ source into 32-bit ARM code or 16-bit Thumb® code.

Typically, you invoke the ARM compiler as follows:

```
armcc [options] ifile_1 ... ifile_n
```

You can specify one or more input files. However, if you specify a dash (-) for an input file, the compiler reads from stdin.

Default behavior

By default, the file suffix you specify changes the configuration assumed by the ARM compiler at start-up. Table 3-1 shows how the compiler startup configuration is adjusted by the filename extension you specify.

<table>
<thead>
<tr>
<th>Filename extension</th>
<th>Instruction set</th>
<th>Source language</th>
</tr>
</thead>
<tbody>
<tr>
<td>.cpp</td>
<td>No adjustment</td>
<td>C++</td>
</tr>
<tr>
<td>.c</td>
<td>No adjustment</td>
<td>No adjustment</td>
</tr>
<tr>
<td>.tc</td>
<td>Thumb</td>
<td>C</td>
</tr>
<tr>
<td>.tcpp</td>
<td>Thumb</td>
<td>C++</td>
</tr>
<tr>
<td>.ac</td>
<td>ARM</td>
<td>C</td>
</tr>
<tr>
<td>.acpp</td>
<td>ARM</td>
<td>C++</td>
</tr>
</tbody>
</table>
Invoking the ARM compiler using older tool names

For backwards compatibility, you can still invoke the ARM compiler using one of the four tool names that were supported in the ARM compilation tools before RVCT v2.1. The startup configuration associated with each of the older tool names is shown in Table 3-2.

Table 3-2 Startup configuration based on old tool names

<table>
<thead>
<tr>
<th>Tool name</th>
<th>Instruction set</th>
<th>Source language</th>
</tr>
</thead>
<tbody>
<tr>
<td>armcc(^a)</td>
<td>ARM</td>
<td>C</td>
</tr>
<tr>
<td>tcc</td>
<td>Thumb</td>
<td>C</td>
</tr>
<tr>
<td>armcpp</td>
<td>ARM</td>
<td>C++</td>
</tr>
<tr>
<td>tcpp</td>
<td>Thumb</td>
<td>C++</td>
</tr>
</tbody>
</table>

\(^a\) This is included for completeness, even though it is the same tool name as the ARM compiler for RVCT v2.1.

Overriding the default behavior

The command-line options shown in Table 3-3 enable you to override the adjustments that the ARM compiler makes based on the filename extension (see Table 3-1 on page 3-2) or the tool name you used to invoke the compiler (see Table 3-2).

Table 3-3 Startup configuration as adjusted by overriding options

<table>
<thead>
<tr>
<th>Command-line option</th>
<th>Instruction set</th>
<th>Source language</th>
</tr>
</thead>
<tbody>
<tr>
<td>--c90</td>
<td>No adjustment</td>
<td>C</td>
</tr>
<tr>
<td>--cpp</td>
<td>No adjustment</td>
<td>C++</td>
</tr>
<tr>
<td>--arm</td>
<td>ARM</td>
<td>No adjustment</td>
</tr>
<tr>
<td>--thumb</td>
<td>Thumb</td>
<td>No adjustment</td>
</tr>
</tbody>
</table>
For example, the following command-line causes the compiler to make determinations as shown in Table 3-4:

tcpp foo.acpp --c90

The configuration that results from these considerations is shown in the Result row at the bottom of the table.

<table>
<thead>
<tr>
<th>Example command component</th>
<th>Description</th>
<th>Instruction set</th>
<th>Source language</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcpp</td>
<td>Tool name</td>
<td>Thumb</td>
<td>C++</td>
</tr>
<tr>
<td>.acpp</td>
<td>Filename extension</td>
<td>ARM</td>
<td>C++</td>
</tr>
<tr>
<td>--c90</td>
<td>Command-line option</td>
<td>No adjustment</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Result</td>
<td>ARM</td>
<td>C</td>
</tr>
</tbody>
</table>

To summarize, the filename extension overrides the default configuration determined by the tool name you use to invoke the ARM compiler, and the command-line option overrides the default configuration determined by the filename extension.

**Building an example**

Sample C source code for a range of applications is installed in `install_directory\RVDS\Examples\...`. Each example is accompanied by a readme.txt file that describes the example code and how to build it.

For example, source code for a simple Dhrystone program is installed in `install_directory\RVDS\Examples\...\dhrystone`. This can be used to measure integer processing performance of a system.

To build the Dhrystone example:

1. Compile the C files with:

   ```bash
   armcc -c -W -g -O3 -Otime --no_inline --no_multifile -DMSC_CLOCK dhry_1.c dhry_2.c
   ``

   where:

   - `-c` Tells the compiler to compile only (not to link).
   - `-W` Tells the compiler to disable all warnings.
-g  Tells the compiler to add debug tables.
-O3  Tells the compiler to generate code with maximum optimizations applied. Used with -g, the debug view might be less satisfactory.
-o time  Tells the compiler to optimize the code for speed (not space).
--no_inline  Tells the compiler to disable function inlining (required by Dhrystone).
--no_multifile  Tells the compiler to disable the Multifile compilation optimization, which is included by default in the -O3 optimization.
-DMSC_CLOCK  Tells the compiler to use the C library function clock() for timing measurements.

To build a Thumb version use:
armcc --thumb ...
where:
--thumb  Tells the compiler to generate Thumb code. (The alternative option, --arm, tells the compiler to generate ARM code, and is the default.)

For full details on the options used here, see the chapter describing how to use the ARM compiler in RealView Compilation Tools v2.1 Compiler and Libraries Guide.

2. Link the image using the command:
armlink dhry_1.o dhry_2.o -o dhrystone.axf --info totals
where:
-o  Specifies the output file as dhrystone.axf.
--info totals  Tells the linker to display totals of the Code and Data sizes for input objects and libraries.

3. Use an ELF/DWARF2 compatible debugger, for example RealView Debugger or AXD, to load and test the image.

See the Readme.txt file that accompanies the example for information on the contents of dhry_1.c and dhry_2.c and how Dhrystone performance is calculated.
Using the ARM linker

The default output from the linker is a non-relocatable image where the code starts at 0x8000 and the data section is placed immediately after the code. You can specify exactly where the code and data sections are located by using linker options or a scatter-loading description file.

For more information see:

- `armlink` command syntax in *RealView Compilation Tools v2.1 Linker and Utilities Guide* for a detailed description of the linker options
- the chapter describing scatter-loading description files in *RealView Compilation Tools v2.1 Linker and Utilities Guide*
- the chapter describing how to develop embedded software in *RealView Compilation Tools v2.1 Developer Guide*.

Using fromELF

An executable image in ELF executable format can be converted to other file formats by using the ARM fromELF utility. For more information, see the chapter describing fromELF in *RealView Compilation Tools v2.1 Linker and Utilities Guide*.

3.1.2 Using the ARM assembler

The basic syntax to use the ARM assembler (armasm) is:

```
armasm inputfile
```

For example, to assemble the code in a file called `myfile.s`, type:

```
armasm --list myfile.lst myfile.s
```

This produces an object file called `myfile.o`, and a listing file called `myfile.lst`.

For full details of the options and syntax, see *RealView Compilation Tools v2.1 Assembler Guide*.

Example 3-1 on page 3-7 shows a small interworking ARM/Thumb assembly language program. You can use it to explore the use of the assembler and linker.

Building the example

To build the example:

1. Enter the code using any text editor and save the file in your current working directory as `addreg.s`. 
2. Assemble the source file using the command:
   armasm --list addreg.lst addreg.s

3. Link the file using the command:
   armlink addreg.o -o addreg

4. Use an ELF/DWARF2 compatible debugger, for example RealView Debugger or AXD, to load and test the image. Step through the program and examine the registers to see how they change (see your debugger documentation for details on how to do this).

For more details on ARM and Thumb assembly language, see RealView Compilation Tools v2.1 Assembler Guide.

---

Example 3-1

```
AREA     AddReg,CODE,READONLY ; Name this block of code.
PRESERVE8 ; Preserve eight-byte alignment.
ENTRY     ; Mark first instruction to call.
main
  ADR r0, ThumbProg + 1    ; Generate branch target address and set bit 0
                          ; hence arrive at target in Thumb state.
  BX r0                         ; Branch and exchange to ThumbProg.
  CODE16        ; Subsequent instructions are Thumb code.
ThumbProg
  MOV r2, #2                       ; Load r2 with value 2.
  MOV r3, #3      ; Load r3 with value 3.
  ADD r2, r2, r3       ; r2 = r2 + r3
  ADR r0, ARMProg              ; Generate branch target address with bit 0 zero.
  BX r0     ; Branch and exchange to ARMProg.
  CODE32     ; Subsequent instructions are ARM code.
ARMProg
  MOV r4, #4
  MOV r5, #5
  ADD r4, r4, r5
  stop MOV r0, #0x18          ; angel_SWIreason_ReportException
  LDR r1, =0x20026       ; ADP_Stopped_ApplicationExit
  SWI 0x0123456    ; ARM semihosting SWI
END                           ; Mark end of this file.
```
3.2 Using ARM libraries

The ISO C and C++ runtime libraries are provided to support compiled C and C++. These are installed in the following subdirectories within `install_directory\RVCT\Data\...\lib`:

- **armlib**: Contains the variants of the ARM C library, the floating-point arithmetic library, and the math library. The accompanying header files are in `install_directory\RVCT\Data\...\include`.

- **cpplib**: Contains the variants of the Rogue Wave C++ library and supporting C++ functions. These functions are collectively referred to as the ARM C++ Libraries. The accompanying header files are installed in `install_directory\RVCT\Data\...\include`.

This section gives an introduction to using the supplied libraries:

- Using the ARM libraries in a semihosted environment
- Using the ARM libraries in a nonsemihosted environment on page 3-9
- Building an application without the ARM libraries on page 3-9.

See the chapter on the C and C++ libraries in *RealView Compilation Tools v2.1 Compiler and Libraries Guide* for more information.

3.2.1 Using the ARM libraries in a semihosted environment

As supplied, the ISO C libraries use the standard ARM semihosted environment to provide facilities such as file input/output. This environment is supported by RealView ICE, *RealView ARMulator® ISS* (RVISS), Multi-ICE®, RealMonitor, and Angel.

If you are developing an application to run in a semihosted environment for debugging, you must have an execution environment that supports the ARM and Thumb semihosting SWIs and has sufficient memory.

The execution environment can be provided by either:

- using the standard semihosting functionality that is present by default in, for example, RealView ICE, Multi-ICE, and RVISS
- implementing your own SWI handler for the semihosting SWI.

You do not have to write any new functions or include files if you are using the default semihosting functionality of the supplied libraries.

See the chapter on semihosting in *RealView Compilation Tools v2.1 Compiler and Libraries Guide* for more information.
3.2.2 Using the ARM libraries in a nonsemihosted environment

If you do not want to use any semihosting functionality, you must ensure that either no calls are made to any function that uses semihosting or that such functions are replaced by your own nonsemihosted functions.

To build an application that does not use semihosting functionality:

1. Create the source files to implement the target-dependent features, for example __user_initial_stackheap().
2. Use #pragma import(__use_no_semihosting_sw) to guard the source.
3. Link the new objects with your application.
4. Use the new configuration when creating the target-dependent application.

You must re-implement functions that the C library uses to insulate itself from target dependencies. For example, if you use printf() you must re-implement fputc(). If you do not use the higher level input/output functions like printf(), you do not have to re-implement the lower level functions like fputc().

If you are building an application for a different execution environment, you can re-implement the target-dependent functions (functions that use the semihosting SWI or that depend on the target memory map). There are no target-dependent functions in the C++ library.

Examples of embedded applications that do not use a hosted environment are included in install_directory\RVDS\Examples\...\emb_sw_dev.

For more information, and examples, see:

- the chapter on C and C++ libraries in RealView Compilation Tools v2.1 Compiler and Libraries Guide
- the chapter describing how to develop embedded software in RealView Compilation Tools v2.1 Developer Guide.

3.2.3 Building an application without the ARM libraries

Creating an application that has a main() function causes the C library initialization functions to be included.

If your application does not have a main() function, the C library is not initialized and the following features are not available to your application:

- software stack checking
- low-level stdio
Creating an Application

- signal-handling functions, signal() and raise() in signal.h
- atexit()
- alloca().

You can create an application that consists of customized startup code, instead of the library initialization code, and still use many of the library functions. You must either:
- avoid functions that require initialization
- provide the initialization and low-level support functions.

These applications will not automatically use the full C runtime environment provided by the C library. Even though you are creating an application without the library, some helper functions from the library must be included. There are also many library functions that can be made available with only minor re-implementations.
3.3 Using your own libraries

The ARM librarian, armar, enables sets of ELF object files to be collected together and maintained in libraries. Such a library can then be passed to armlink in place of several object files. However, linking with an object library file does not necessarily produce the same results as linking with all object files collected into the object library file. This is because armlink processes the input list and libraries differently:

- each object file in the input list appears in the output unconditionally, although unused areas are eliminated if the armlink --remove option is specified
- a member of a library file is included in the output only if it is referred to by an object file or a previously processed library file.

To create a new library called my_lib and add all object files in the current directory, type:

armar --create my_lib *.o

To delete all objects from the library that have a name starting with sys_, type:

armar -d my_lib sys_*

To replace, or add, three objects in the library with the version located in the current directory, type:

armar -r my_lib obj1.o obj2.o obj3.o

For more information on armar, see the chapter on creating and using libraries in *RealView Compilation Tools v2.1 Linker and Utilities Guide*.

--- Note ---

The ARM libraries must not be modified. If you want to create a new implementation of a library function, place the new function in an object file or your own library. Include your object or library when you link the application. Your version of the function is used instead of the standard library version.
Glossary

The items in this glossary are listed in alphabetical order, with any symbols and numerics appearing at the end.

AAPCS
See Procedure Call Standard for the ARM Architecture.

ABI for the ARM Architecture (base standard) (BSABI)
The ABI for the ARM Architecture is a collection of specifications, some open and some specific to ARM architecture, that regulate the inter-operation of binary code in a range of ARM-based execution environments. The base standard specifies those aspects of code generation that must be standardized to support inter-operation and is aimed at authors and vendors of C and C++ compilers, linkers, and runtime libraries.

ADS
See ARM Developer Suite.

Angel
A debug monitor that enables you to develop and debug applications running on hardware that is based on an ARM core. Angel can debug applications running in either ARM state or Thumb state.

APCS
ARM Procedure Call Standard.

API
See Application Programming Interface.

Application Programming Interface
The syntax of the functions and procedures within a module or library.
ARM Developer Suite (ADS)  
A suite of software development applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors. ADS is superseded by RealView Developer Suite (RVDS).

See also RealView Developer Suite.

ARM instruction  
A word that specifies an operation for an ARM processor to perform. ARM instructions must be word-aligned.

ARM state  
A processor that is executing ARM (32-bit) instructions is operating in ARM state.

See also Thumb state.

armar  
The ARM librarian.

armasm  
The ARM assembler.

armcc  
The ARM C compiler.

armlink  
The ARM linker.

Big-endian  
Memory organization in which the least significant byte of a word is at a higher address than the most significant byte.

Breakpoint  
A user defined point at which execution stops so that a debugger can examine the state of memory and registers.

BSABI  
See ABI for the ARM Architecture (base standard).

Byte  
An 8-bit data item.

Class  
A C++ class involved in the image.

CLI  
C Language Interface or Command-line Interface.

Command-line Interface  
You can operate any ARM debugger by issuing commands in response to command-line prompts. This is the only way of operating armsd, but AXD and RealView Debugger both offer a graphical user interface in addition. 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.

Coprocessor  
An additional processor used for certain operations. Usually used for floating-point math calculations, signal processing, or memory management.

CPU  
Central Processor Unit.
Current Program Status Register (CPSR)

See Program Status Register.

Debug With Arbitrary Record Format (DWARF)

ARM code generation tools generate debug information in DWARF2 format.

Deprecated

A deprecated option or feature is one that you are strongly discouraged from using. Deprecated options and features will not be supported in future versions of the product.

DLL

See Dynamic Linked Library.

DWARF

See Debug With Arbitrary Record Format.

Dynamic Linked Library (DLL)

A collection of programs, any of which can be called when required by an executing program. A small program that helps a larger program communicate with a device such as a printer or keyboard is often packaged as a DLL.

ELF

See Executable and Linking Format.

Embedded

Used to refer to either:

- applications that are developed as firmware
- assembler functions placed out-of-line in a C or C++ program.

See also Inline.

Exception

Handles an event. For example, an exception could handle an external interrupt, or an undefined instruction.

Executable and Linking Format (ELF)

The industry standard binary file format used by RealView Compilation Tools. ELF object format is produced by the ARM object producing tools such as armcc and armasm. The ARM linker accepts ELF object files and can output either an ELF executable file, or partially linked ELF object.

FIQ

Fast Interrupt.

Flash memory

Nonvolatile memory that is often used to hold application code.

Floating Point Unit (FPU)

A hardware unit dedicated to performing arithmetic operations on floating-point values.

Floating-point

Convention used to represent real (as opposed to integer) numeric values. Several such conventions exist, trading storage space required against numerical precision.

FP

See Floating-point

FPU

See Floating Point Unit.
fromELF The ARM image conversion utility. This accepts ELF format input files and converts them to a variety of output formats. fromELF can also generate text information about the input image, such as code and data size.

GCC See GNU Compiler Collection.

Global variables Variables with global scope within the image.

GNU Compiler Collection (GCC) The GNU Compiler Collection contains front ends for C, C++, and supporting libraries.

GUI Graphical User Interface.

Heap The portion of computer memory that can be used for creating new variables.

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.

Immediate values Values that are encoded directly in the instruction and used as numeric data when the instruction is executed. Many ARM and Thumb instructions enable small numeric values to be encoded as immediate values within the instruction that operates on them.

Inline Used to refer to either:
- 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.

International Standards Organization (ISO) An organization that specifies standards for, among other things, computer software. This supersedes the American National Standards Institute.

Interworking A method of working that enables branches between ARM and Thumb code.

IRQ Interrupt Request.


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.
| **Little-endian** | Memory organization in which most significant byte of a word is at a higher address than the least significant byte. |
| **Local** | An object that is only accessible to the subroutine that created it. |
| **Memory Management Unit (MMU)** | Enables detailed control of a memory system. Most of the control is provided through translation tables held in memory. |
| **MMU** | *See Memory Management Unit.* |
| **Multi-ICE** | A JTAG-based tool for debugging embedded systems. |
| **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.* |
| **PI** | Position-Independent. |
| **Procedure Call Standard for the ARM Architecture (AAPCS)** | *Procedure Call Standard for the ARM Architecture* defines how registers and the stack will be used for subroutine calls. |
| **Processor core** | The part of a microprocessor that reads instructions from memory and executes them, including the instruction fetch unit, arithmetic and logic unit and the register bank. It excludes optional coprocessors, caches, and the memory management unit. |
| **Program Counter (PC)** | Integer register R15. |
| **Program Status Register (PSR)** | Contains some information about the current program and some information about the current processor. Also referred to as Current PSR (CPSR), to emphasize the distinction between it and the Saved PSR (SPSR). The SPSR holds the value the PSR had when the current function was called, and which will be restored when control is returned. An Enhanced Program Status Register (EPSR) contains an additional bit (the Q bit, signifying saturation) used by some ARM processors, including the ARM9E. |
| **RAM** | Random Access Memory. |
| **RDI** | *See Remote Debug Interface.* |
| **Read-Only Position Independent (ROPI)** | Code or read-only data that can be placed at any address. |
| **Read/Write Position Independent (RWPI)** | Read/write data addresses that can be changed at runtime. |
RealView ARMulator ISS (RVISS)
The most recent version of the ARM simulator, RealView ARMulator ISS is supplied with RealView Developer Suite. It communicates with a debug target using RV-msg, through the RealView Connection Broker interface, and RDI.

See also RDI and RealView Connection Broker.

RealView Compilation Tools (RVCT)
RealView Compilation Tools is a suite of tools, together with supporting documentation and examples, that enables you to write and build applications for the ARM family of RISC processors.

RealView Connection Broker
RealView Connection Broker is an execution vehicle that enables you to connect to simulator targets on your local system, or on a remote system. It also enables you to make multiple connections to the simulator.

See also RealView ARMulator ISS.

RealView Developer Suite (RVDS)
The latest suite of software development applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors.

RealView ICE (RVI)
A JTAG-based debug solution to debug software running on ARM processors.

Regions
A contiguous sequence of one to three output sections (RO, RW, and ZI) in an image.

Register
A processor register.

Remote Debug Interface (RDI)
The Remote Debug Interface (RDI) is an ARM standard procedural interface between a debugger and the debug agent. RDI gives the debugger a uniform way to communicate with:

- a simulator running on the host (for example, RVISS)
- a debug monitor running on hardware that is based on an ARM core accessed through a communication link (for example, Angel)
- a debug agent controlling an ARM processor through hardware debug support (for example, RealView ICE or Multi-ICE).

Retargeting
The process of moving code designed for one execution environment to a new execution environment.

RISC
Reduced Instruction Set Computer.

ROM
Read Only Memory.

ROPI
See Read Only Position Independent.
**Rounding modes**  Specify how the exact result of a floating-point operation is rounded to a value that is representable in the destination format.

**RVCT**  *See* RealView Compilation Tools.

**RVDS**  *See* RealView Developer Suite.

**RWPI**  *See* Read Write Position Independent.

**Saved Program Status Register (SPSR)**  *See* Program Status Register.

**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.

**Script**  A file specifying a sequence of debugger commands that you can submit to the command-line interface using the *obey* command. This saves you from having to enter the commands individually, and is particularly helpful when you have to issue a sequence of commands repeatedly.

**SDT**  *See* Software Development Toolkit.

**Sections**  A block of software code or data for an Image.

*See also* Input section and Output section.

**Semihosting**  A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather than attempting to support the I/O itself.

**Software Development Toolkit (SDT)**

Software Development Toolkit (SDT) is an ARM product, now superseded by ADS and RVCT.

**Software Interrupt (SWI)**  An instruction that causes the processor to call a programmer-specified subroutine. Used by the ARM standard C library to handle semihosting.

**Source File**  A file that is processed as part of the image building process, for example a C or C++ file, or an assembler file.

**SP (Stack Pointer)**  Integer register R13.

**SPSR**  Saved Program Status Register.

*See also* Program Status Register.
Glossary

**Stack**
The portion of memory that is used to record the return address of code that calls a subroutine. The stack can also be used for parameters and temporary variables.

**SWI**
*See* Software Interrupt.

**Target**
The target hardware, including processor, memory, and peripherals, real or simulated, on which the target application is running.

**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.

**Thumb state**
A processor that is executing Thumb (16-bit) instructions is operating in Thumb state.
*See also* ARM state.

**Translation tables**
Tables held in memory that define the properties of memory areas of various sizes from 1KB to 1MB.

**Unsigned data types**
Represent a non-negative integer in the range 0 to +2^N-1, using normal binary format.

**Variable**
A named memory location of an appropriate size to hold a specific data item.

**Vector Floating Point (VFP)**
A standard for floating-point coprocessors where several data values can be processed by a single instruction.

**VFP**
*See* Vector Floating Point.

**Word**
Value held in four contiguous bytes. A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
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

ABI 1-3
ar 1-3
ARM
  non-semihosting 3-9
  semihosting 3-8
ARM C Library
  armlib 3-8
ARM Librarian 3-11
ARM mode 2-4
armar 1-3, 3-11
armasm 1-2
  example 3-6
armcc 1-2, 3-2
  example 3-4
armlib 3-8
armlink 1-2
  example 3-5
  output defaults 3-6
  scatter-loading description files 3-6
ARMulator
  RealView ARMulator ISS 1-4

Assembler
  ARM 1-2
  example 3-6
  using from the command line 3-6

B

Books
  Assembler Guide vii
  Compilers and Libraries Guide vii
  Developer Guide vii
  Linker and Utilities Guide vii
  BSABI 1-3

C

Command line
  compiler example 3-4
  compiling from 3-2
  using the assembler from 3-6

Compiler
  ARM 1-2, 3-4
  default behavior 3-2
  example 3-4
  overriding default behavior 3-3
  Thumb 1-2, 3-4
  using from the command line 3-2
  using old tool names 3-3
  cpplib 3-8
  C++ library
  Rogue Wave 1-2

D

Dhrystone
  example 3-4

Documentation
  DynaText 1-5
  HTML 1-6
  online 1-5
  PDF 1-5
  Red Hat Linux 1-5
  Rogue Wave 1-6
Index

Sun Solaris 1-5
Windows 1-5
DWARF2 1-3

E
EDG
front end 2-4
ELF 1-3
file format 1-3
format 1-3
fromELF 1-3, 3-6

Examples
compiler 3-4
Dhrystone 3-4
embedded 3-4
finding 3-4
Thumb 3-4

F
fromELF 1-3, 3-6

G
GNU extensions
ARM mode 2-4
GNU mode 2-4
GNU mode 2-4

I
Installation directory
default location vi
Interworking
element 3-6
ISO/IEC 14822
1998 C++ 1-3
ISO/IEC 9899
1990 C 1-3

L
Librarian 3-11

Libraries
ARM 3-8
arm 3-11
armlib 3-8
C support 1-3
cpplib 3-8
creating functions 3-11
custom 3-11
C++ runtime support 1-2
location 3-8
nonhosted environment 3-9
non-semihosting 3-9
programming without using 3-9
semihosting 3-8
semihosting dependencies 3-9
using customized startup code 3-10

Linker
ARM 1-2
development tool 1-2
output defaults 3-6
scatter-loading description files 3-6

N
Non-semihosting 3-9
Normative Addendum 1 1-3

O
Object files
adding to library 3-11
deleting from library 3-11
producing with assembler 3-6
replacing in library 3-11

R
RealView ARMulator ISS 1-4
RealView Connection Broker 1-4
Remote Debug Interface
RDI 1-4
Rogue Wave
C++ library 1-2
documentation 1-6
Rogue Wave C++ Library
cpplib 3-8

RVCT
C library 1-3
components 1-2
C++ runtime libraries 1-2
deprecated features in v2.1 2-2
development tools 1-2
documentation 1-5
new features in v2.0 2-7
new features in v2.1 2-3
new features in v2.1 (overview) 2-2
supported standards 1-3
supporting software 1-4
RV-msg 1-4

S
Scatter-loading
description files 3-6
Semihosting 3-8
non-semihosting 3-9
 Standards
supported by RVCT 1-3

T
Thumb 1-2
Tools 1-2

Copyright © 2002-2004 ARM Limited. All rights reserved.