RealView® Compilation Tools

Version 2.2

Essentials Guide

ARM®

ARM DUI 0202F

RealView Compilation Tools
Essentials Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
August 2002	A	Release 1.2
January 2003	B	Release 2.0
September 2003	C	Release 2.0.1 for RVDS v2.0
January 2004	D	Release 2.1 for RVDS v2.1
December 2004	E	Release 2.2 for RVDS v2.2
May 2005	F	Release 2.2 for RVDS v2.2 SP1

Proprietary Notice

Confidentiality Status

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

This preface introduces the *RealView Compilation Tools Essentials Guide*. It contains the following sections:

About this book

This book provides an overview of the *RealView® Compilation Tools* (RVCT) 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 and appendixes:

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 the new release of RVCT and the previous release.

Chapter 3 *Creating an Application*

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

Appendix A *About Previous Releases*

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

Further reading

This section lists publications from 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 v2.2 Developer Guide* (ARM DUI 0203). This book provides tutorial information on writing code targeted at the ARM family of processors.
- *RealView Compilation Tools v2.2 Assembler Guide* (ARM DUI 0204). This book provides reference and tutorial information on the ARM assembler.

- *RealView Compilation Tools v2.2 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 v2.2 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 full information about the base standard, 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 Architecture Reference Manual* (ARM DDI 0100)
- *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 Steve Furber, *ARM system-on-chip architecture* (2nd edition, 2000). Addison Wesley, ISBN 0-201-67519-6.

Feedback

ARM Limited welcomes feedback on both RealView Compilation Tools and its documentation.

Feedback on the RealView Compilation Tools

If you have any problems with RVCT, 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® Compilation Tools* (RVCT) 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 1-7.

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 *Reduced Instruction Set Computing* (RISC) processors.

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

1.1.1 Components of RVCT

For information about RVCT see:

- *Development tools*
- *Standards compliance* on page 1-4
- *Compliance with the ABI for the ARM Architecture (base standard)* on page 1-4
- *Supporting software* on page 1-6
- *Code examples* on page 1-6.

Development tools

The following development tools are provided when you install RVCT:

armcc The ARM and Thumb® C and C++ compiler. It compiles either:

- ISO C:1990 source
- ISO C++:1998 source

into:

- 32-bit ARM code
- 16-bit Thumb code
- 16/32-bit Thumb-2 code.

Introduced in RVCT v2.2, the Thumb-2 instruction set includes older 16-bit Thumb instructions as a subset. Thumb-2 introduces many new 32-bit instructions and some new 16-bit instructions. On those processors that support it, Thumb-2 gives near ARM performance together with code size that is the same as older versions of Thumb.

armasm The ARM and Thumb assembler. This assembles ARM assembly language and Thumb/Thumb-2 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\...`.

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 I/O).

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.

Note

All RVCT tools are 32-bit applications and, therefore, have an address (memory accessed) limit of 4GB. Sun Solaris machines are all 64-bit but they will hit memory problems when building large images that extend over the 4GB boundary. This generates a warning message (L6000U) to indicate that there is not enough memory. This might cause confusion since sufficient memory is available but the application cannot access it.

Standards compliance

RVCT conforms to the following standards. In each case, the level of compliance is noted:

ar `armar` produces, and `armlink` consumes, UNIX-style object code archives. `armar` can list and extract most `ar`-format object code archives, and `armlink` can use an `ar`-format archive created by another archive utility provided that it contains a symbol table member.

DWARF2 DWARF2 debug tables are supported by all the tools in the RVCT suite, and by ELF/DWARF2 compatible debuggers from ARM, for example, RealView Debugger and *ARM eXtended Debugger* (AXD).

Note

The DWARF2 standard is ambiguous in some areas (such as debug frame data) so there is no guarantee that third-party debuggers can consume the DWARF2 produced by ARM or that ARM debuggers can consume the DWARF2 produced by third-party tools.

DWARF3 Initial support for DWARF3 (Draft Standard 9) debug tables is provided by all the tools in the RVCT suite, and by RealView Debugger v1.8 (and above).

ISO C The ARM compiler accepts *ISO/IEC 9899:1990 C*, including the *Normative Addendum 1* from 1994 (except file I/O), 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 relocatable and executable files in ELF format. The fromelf utility can translate ELF files into other formats.

Compliance with the ABI for the ARM Architecture (base standard)

The *Application Binary Interface* (ABI) for the ARM Architecture is a collection of standards, some open and some specific to the ARM architecture, that regulate the inter-operation of binary code and development tools in ARM-based execution environments ranging from bare metal to major operating systems such as ARM Linux.

By conforming to this standard, ARM and Thumb objects and libraries from different producers can work together.

The *ABI for the ARM Architecture (base standard)* [BSABI] consists of a family of specifications including:

AAPCS *Procedure Call Standard for the ARM Architecture*. Governs the exchange of control and data between functions at runtime. There is a variant of the AAPCS for each of the major execution environment types supported by RVCT.

CPPABI *C++ ABI for the ARM Architecture*. Builds on the generic C++ ABI (originally developed for IA-64) to govern interworking between independent C++ compilers.

EHABI *Exception Handling ABI for the ARM Architecture.* Defines both the language-independent and C++-specific aspects of how exceptions are thrown and handled.

AAELF *ELF for the ARM Architecture*. Builds on the generic ELF standard to govern the exchange of linkable and executable files between producers and consumers.

AADWARF *DWARF for the ARM Architecture.* This ABI uses DWARF 3.0 to govern the exchange of debugging data between object producers and debuggers.

RTABI *Runtime ABI for the ARM Architecture*. Governs what independently produced objects can assume of their execution environments by way of floating-point and compiler helper function support.

CLIBABI *C Library ABI for the ARM Architecture*. Defines an ABI to the ISO C:1990 library.

BPABI *Base Platform ABI for the ARM Architecture*. Governs the format and content of executable and shared object files generated by static linkers. Supports platform-specific executable files using post linking. Provides a base standard that is used to derive a platform ABI.

For more information on the base standard, software interfaces, and standards supported by ARM, see *install_directory*`\Documentation\Specifications\...`.

For details of the latest published versions, see `http://www.arm.com`.

If you are upgrading to the latest release of RVCT from a previous release, ensure that you are using the most recent versions of the ARM specifications.

Supporting software

To debug your programs under simulation, or on hardware that is based on an ARM core, use a suitable debugger, for example, ARM RealView Debugger v1.8 (compatible with ELF, DWARF2, or DWARF3 as produced by GCC 3.4 or RVCT v2.2) or AXD (ELF/DWARF2).

To debug your programs under simulation, use the RealView ARMulator® ISS (RVISS) supporting software. RVISS is an *Instruction Set Simulator* (ISS) that is supplied with RealView Developer Suite. It communicates with a debugger, for example, ARM RealView Debugger, and can run on the same host computer or on a system remote from that running the debugger. For more details, see *RealView ARMulator ISS User Guide*.

Code examples

This book references examples provided with RealView Developer Suite in the main examples directory *`install_directory`*`\RVDS\Examples`. See *RealView Developer Suite Getting Started Guide* for a summary of the examples provided.

1.2 Getting more information online

Depending on your installation, the full documentation suite is available online as PDF and DynaText for Windows and Sun Solaris, and as PDF for Red Hat Linux.

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:

- *RVCT on Windows*
- *RVCT on Sun Solaris and Red Hat Linux*
- *Rogue Wave documentation* on page 1-8.

1.2.1 RVCT on Windows

Select **Programs → ARM** from the Windows **Start** menu. From here select either:

- **RealView Developer Suite v2.2 → PDF Documentation** to view the PDF files.
- **DynaText Documentation** to view the DynaText files.

Unless you change the defaults, the PDF and DynaText files are installed in *install_directory*\Documentation\RVCT\...\windows.

1.2.2 RVCT on Sun Solaris and Red Hat Linux

The full documentation suite is available as PDF and DynaText files for Sun Solaris, and as PDF on Red Hat Linux. If 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 *install_directory*/Documentation/RVCT/.../unix.

Note

DynaText files are not installed 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).

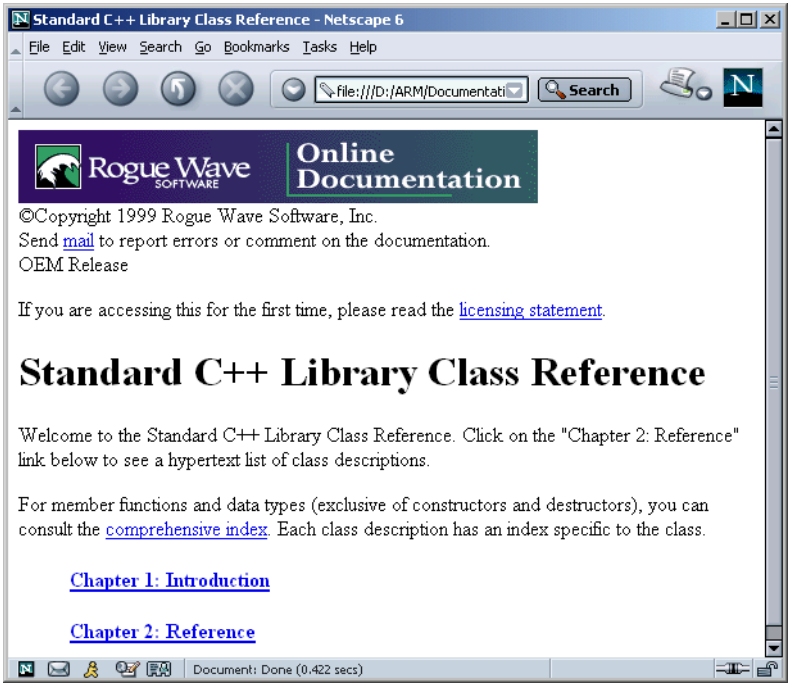

Figure 1-1 HTML browser

Chapter 2
Differences

This chapter describes the major differences between the latest release of *RealView® Compilation Tools* (RVCT) and the previous release (RVCT v2.1).

It contains the following sections:

- *RVCT v2.2 SP1 overview* on page 2-2
- *RVCT v2.2 overview* on page 2-4
- *Differences between RVCT v2.2 and RVCT v2.1* on page 2-8.

For information about previous releases of RVCT, see:

- Appendix A *About Previous Releases*
- Appendix E in *RealView Compilation Tools v2.2 Compiler and Libraries Guide.*

2.1 RVCT v2.2 SP1 overview

This section describes the differences between RVCT v2.2 delivered with RVDS v2.2 Service Pack 1 (SP1) and the previous release.

The differences are:

- RVCT v2.2 includes support for the ARM1136J(F)-S-rev-1 core. To see a full list of supported cores use:

  ```
  armcc --cpu list
  ```

- The ARM linker offers new options when building shared objects, that is, use:

  — `--pt_arm_exidx` to create a `PT_ARM_EXIDX` program header that describes the location of the exception tables. The linker assumes that a shared object containing a `PT_ARM_EXIDX` program header might throw an exception.

  — `--force_so_throw` to force the linker to assume that all shared objects might throw an exception.

  For full details of these command-line options, see the chapter describing System V shared libraries in *RealView Compilation Tools v2.2 Linker and Utilities Guide*.

- `fromelf` adds support for a `--no_comment_section` option, to remove the .comment section from ELF output files.

- The storage type selection of the compiler option `--enum_is_int` has changed. If the range of the enumerator is greater than the range of a **`signed int`**, but is within the range of an **`unsigned int`**, the storage type of the **`enum`** is now **`unsigned int`**.

  For example:

  ```
  enum E { k = 0x80000000 };
  ```

  is treated as an **`unsigned int`**. In previous versions of RVCT a warning is generated.

- The keywords **`__restrict`** and **`__restrict__`** are supported as synonyms for **`restrict`**. In this release, both keywords are available in all modes (not just with `--restrict`).

- The `deprecated` function attribute is now supported in both GNU mode and ARM mode:

  ```
  int Function_Attributes_deprecated_0(int b) __attribute__ ((deprecated));
  ```

- The `SWI` instruction has been renamed to `SVC` (SuperVisor Call).

  In this release of RVCT, instructions disassemble to `SVC`, with a comment to say that this was formerly `SWI`. For example, `fromelf --text -c` produces:


```
...
0x00000fbc:    e3a00010    ....    MOV      r0,#0x10
0x00000fc0:    ef123456    V4..    SVC      0x123456 ; formerly SWI
...
```

2.2 RVCT v2.2 overview

The most important differences between RVCT v2.2 and RVCT v2.1 are:

- RVCT v2.2 supports the new Thumb®-2 instruction set. Thumb-2 introduces many new 32-bit instructions, and some new 16-bit instructions. Thumb-2 gives near ARM performance together with code size that is the same as older versions of Thumb on those processors that support it, such as the ARM1156T2(F)-S.

  The Thumb-2 instruction set includes older 16-bit Thumb instructions as a subset.

- RVCT v2.2 includes support for new ARMv6 cores, for example, the ARM1176JZ(F)-S, incorporating ARM TrustZone™ technology-optimized software, the ARM968EJ-S, the ARM1156T2(F)-S, and the ARM MPCore™.

  The RVCT v2.2 assembler provides support for MPCore instructions.

- RVCT v2.2 includes a substantially upgraded assembler, while continuing to support the old version to enable the assembly of old code.

- RVCT v2.2 is fully compliant with the *Base Platform ABI for the ARM Architecture* [BPABI], so giving support for a range of operating systems, for example, ARM Linux and Symbian OS.

- RVCT v2.2 provides initial support for DWARF3 (Draft Standard 9) debug tables, as described in the *ABI for the ARM Architecture (base standard)* [BSABI]. A new command-line option, `--dwarf3`, is now available to specify this format when compiling code.

- The command-line option `-g` switches on the generation of debug tables for the current compilation. Optimization options are specified by `-O`*num*. By default, using the `-g` option does not affect the optimization setting.

  This is a change in behavior for RVCT v2.2 (`-g` alone was equivalent to `-g -O0` in RVCT v2.1) and might change in a future release. Apart from this new default, there is no change in the behavior of the optimization options, that is, `-O`*num*, `-Ospace`, or `-Otime`.

- RVCT v2.2 supports the command-line option `--apcs /fpic` to compile code that is compatible with System V shared libraries. Use this option to generate read-only position-independent code where relative address references are independent of the location where your program is loaded.

- The ARM linker supports building, and linking against, shared libraries. New command-line options are available to build SVr4 executable files and shared objects, and to specify how code is generated.

- The ARM linker conforms to the *Base Platform ABI for the ARM Architecture* [BPABI] and supports the GNU-extended symbol versioning model.
- The ARM implementation of floating-point computations has been changed to provide improved support for C99 functions. Where this changes behavior significantly, a *compatibility mode* has been introduced to aid developers to migrate code to use the new features. See *Changes to library support* on page 2-10 for more information.
- The ARM compiler C implementation has been changed so that it now issues a Warning for out-of-range enumerator values. Such values are treated the same way as in C++. This means the size of **`enum`** types with out-of-range values might be different in C when you upgrade to the latest release of RVCT.

  There is no change if you are using C++, or the command-line options `--enum_is_int`, `--strict`, or `--strict_warnings`. See *Changes to the ARM compiler* on page 2-8 for more information.
- The ARM libraries have been enhanced to provide improved support for multithreading. This is designed to help developers who are working with RTOS-based systems.
- The ARM compiler offers new options to provide greater control of how dynamic symbols are exported.
- The ARM linker has global visibility of all your program code and so can now perform some branch optimizations that are not available to other components of RVCT.
- Two new command-line options are available to handle tail calling sections to optimize the branch instruction at the end of the section.
- Options that were deprecated in RVCT v2.1 are now obsolete in v2.2 (see *Obsolete features* on page 2-6).
- Some options that were supported in RVCT v2.1 are now deprecated in v2.2 (see *Deprecated features* on page 2-7).

See *Differences between RVCT v2.2 and RVCT v2.1* on page 2-8 for more details on new features, and changes in behavior, outlined in this section.

2.2.1 Obsolete features

Be aware of the following differences in RVCT v2.2:

- All features and options that were deprecated in RVCT v2.1 are obsolete in RVCT v2.2, that is:
  - Legacy *Software Development Toolkit* (SDT) formats such as *ARM Object Format* (AOF) and libraries in *ARM Library Format* (ALF) format.
  - The use of single dashes for keywords, for example, `armlink -help`.
  - The compiler options `-ansi` and `-ansic`.
- Old compiler option names that were deprecated in RVCT v2.1 are obsolete in RVCT v2.2, for example, `-fy`, `-fd`, `-Ec`, and `-zo`,

  The compiler issues a warning where a preferred option name exists, for example:

  ```
  armcc -zo
  Warning: X0010W: Old syntax, please use '--split_sections'.
  ```

  See Appendix E in *RealView Compilation Tools v2.2 Compiler and Libraries Guide* for a list.
- Selected older ARM processors and architectures are obsolete:
  - ARM6
  - ARMv3 and ARMv3M.

  To see a full list of supported cores use:

  ```
  armcc --cpu list
  ```
- Selected older floating-point architectures are obsolete:
  - VFPv1 (the default is VFPv2)
  - FPA
  - Soft FPA.

  Unless specified, the default is Soft VFP.

  To see a full list of supported floating-point architectures use:

  ```
  armcc --fpu list
  ```
- The following compiler options are obsolete:
  - `--fpu fpa`, `--fpu softfpa`, and `--fpu vfpv1`
  - `--fa`
  - `--cpu 3` and `--cpu 3M`
  - `-Oldrd` and `-Ono_ldrd`
  - `-W`*`letter`* and `-E`*`letter`*.

- Synonyms for the SXT and UXT instructions (sign extend or zero extend) are not supported.
- Use of FPA registers f0-f7 and F0-F7 is obsolete.
- The use of the environment variable RVCT21_CLWARN to warn against deprecated options is not supported.

2.2.2 Deprecated features

Be aware of the following differences in RVCT v2.2:

- The compiler supports the option --apcs /adsabi to compile code that is compatible with the old *ARM Developer Suite*™ (ADS) *Application Binary Interface* (ABI). This is deprecated and will be removed in a future release.
- The ARM linker and fromelf accept two forms of negated option, for example, --no_debug and --nodebug. However, the non-standard form, for example, --nodebug, is deprecated and will not be supported in the future. A warning is issued if you use the deprecated syntax.
- The C++ configuration option --dll_vtbl has been replaced by the new option --export_all_vtbl. The option --dll_vtbl is deprecated and will not be supported in the future.
- The RVCT assembler supports two forms of the Load Register EXclusive instruction:
  - LDREX{B|D|H}{*cond*} *Rd*, [*Rn*]
  - LDR{B|D|H}EX{*cond*} *Rd*, [*Rn*]

  The second is deprecated and will be removed in a future release.

  The disassembler supports only the first form.
- The RVCT assembler supports two forms of the Store Register EXclusive instruction:
  - STREX{B|D|H}{*cond*} *Rd*, [*Rn*]
  - STR{B|D|H}EX{*cond*} *Rd*, [*Rn*]

  The second is deprecated and will be removed in a future release.

  The disassembler supports only the first form.

2.3 Differences between RVCT v2.2 and RVCT v2.1

RVCT v2.2 overview on page 2-4 describes the most important differences between RVCT v2.2 and RVCT v2.1. This section describes other differences between RVCT v2.2 and RVCT v2.1 in more detail, and includes:

- *Changes to the ARM compiler*
- *Changes to library support* on page 2-10
- *Changes to the ARM linker* on page 2-12
- *Changes to the ARM assembler* on page 2-14
- *Changes to the fromelf utility* on page 2-15.

2.3.1 Changes to the ARM compiler

Be aware of the following differences in the compiler in RVCT v2.2:

- The ARM compiler offers new options to provide greater control of how dynamic symbols are exported, that is, use:
  - `--export_all_vtbl` to export all virtual table functions and RTTI for classes with a key function
  - `--export_defs_implicitly` to export definitions where the prototype was marked `dllimport`.
- The ARM compiler offers new options to give symbol visibility when building shared objects or DLLs, that is, use:
  - `--dllexport_all` to provide dynamic visibility of all global symbols unless specified otherwise
  - `--no_hide_all` to export all `extern` definitions and import all undefined references.
- The new `__swi_indirect_r7` behaves like `__swi_indirect_r12` except that it uses `r7` instead of `r12`. Thumb applications on ARM Linux use `__swi_indirect_r7` to make kernel syscalls.
- The behavior of `--force_new_nothrow` has been enlarged so that it causes any overloaded global `operator new` to be treated as `throw()`.
- The ARM compiler C implementation has been changed so that it now issues a Warning for out-of-range enumerator values.

  In strict C, enumerator values must be representable as **`int`**s, for example, they must be in the range -2147483648 to +2147483647 (inclusive). In previous releases of RVCT out-of-range values were cast to **`int`** without a warning (unless you specified the `--strict` option).

In RVCT v2.2, such values are treated the same way as in C++, that is, they are treated as **unsigned int**, **long long**, or **unsigned long long**. This means the size of **enum** types with out-of-range values might be different in C if you are using the latest release of RVCT. For example:

```
enum E1 { k1 = 0xffffffff };  /* value == 2147483648u; out-of-range in C */
    /* C: before 2.2: sizeof(enum E1) == 1 */
    /* C:        2.2: sizeof(enum E1) == 4 */
    /* C++:      all: sizeof(enum E1) == 4 */
enum E2 { k2 = (int)0xffffffff };  /* value == -1; in range */
    /* C: before 2.2: sizeof(enum E1) == 1 */
    /* C:        2.2: sizeof(enum E1) == 1 */
    /* C++:      all: sizeof(enum E1) == 1 */
enum E3 { k3 = -1, k4 = 0xffffffff };  /* value == 2147483648u; out-of-range in C */
    /* C: before 2.2: sizeof(enum E1) == 1 */
    /* C:        2.2: sizeof(enum E1) == 8; use long long */
    /* C++:      all: sizeof(enum E1) == 8; use long long */
```

To ensure that out-of-range Warnings are reported, use the following command to change them into Errors:

```
armcc --diag_error 66 ...
```

There is no change if you are using C++, or the command-line options `--enum_is_int`, `--strict`, or `--strict_warnings`.

- The ARM compiler now supports the use of the `__attribute__` keyword in both ARM and GNU mode.
- The ARM compiler now supports a new variable attribute, `zero_init` or `__zero_init__`, to specify that a variable with no initializer is placed in a ZI data section.
- The ARM compiler provides a new option, `--dwarf3`, to specify DWARF3 standard debug tables to describe ARM or Thumb programs when compiling code written in C or C++. If you do not specify a format, the compiler assumes DWARF2.
- The ARM compiler support a new environment variable `RVCT22_CCOPT` that enables you to specify command-line options to the compiler.
- RVCT v2.2 supports the command-line option `--apcs /fpic` to compile code that is compatible with System V shared libraries. Use this option to generate read-only position-independent code where relative address references are independent of the location where your program is loaded.

- The ARM compiler provides new intrinsics to control interrupt handling:
  - `__enable_irq()` and `__disable_irq()`
  - `__enable_fiq()` and `__disable_fiq()`.
- The ARM compiler provides new intrinsics to control optimization:
  - `__schedule_barrier()`
  - `__force_stores()`
  - `__memory_changed()`.
- The ARM compiler includes new options to enable or disable implicit determination, from context, whether a template parameter dependent name is a type or nontype, that is, use:
  - `--implicit_typename`
  - `--no_implicit_typename`.

  The default is `--no_implicit_typename`.
- When you invoke the compiler, you can use the `--show_cmdline` option to see how the compiler has processed the command line. The commands are shown normalized, and the contents of any via files are expanded. However, using this option does not trap accidental errors in your command line.
- The ARM compiler is no longer guaranteed to be ISO C and C++ standard-compliant with respect to floating-point arithmetic at optimization level `-O3`. You must use the `--fpmode=std` option to ensure ISO C and C++ compliance.
- The ARM compiler now places **`const volatile`** (and **`volatile const`**) data in RW or ZI sections, as mandated by the ISO C standard. Previously such data was placed in RO sections. This might affect your code if you have made assumptions about the placement of data by the compiler.

2.3.2 Changes to library support

Be aware of the following differences in library support in RVCT v2.2:

- The ARM implementation of floating-point computations has been changed to provide improved support for C99 functions. Where this changes behavior significantly, a *compatibility mode* has been introduced to aid developers to migrate code to use the new features. C99 functions that are new in fplib (or where behavior has changed) are:
  - `ilogb`, `ilogbf`, `ilogbl`
  - `logb`, `logbf`, `logbl`
  - `scalbn`, `scalbnf`, `scalbnl`, `scalbln`, `scalblnf`, `scalblnl`
  - `nextafter`, `nextafterf`, `nextafterl`, `nexttoward`, `nexttowardf`, `nexttowardl`.

C99 functions that are new in mathlib (or where behavior has changed) are:

— `fpclassify` and `signbit`

— `isfinite`, `isinf`, `isnan`, and `isnormal`

— `copysign`, `copysignf`

— `isgreater`, `isgreaterequal`, `isless`, `islessequal`, `islessgreater`, and `isunordered`.

To help you to port your code, the new compatibility mode emulates the previous behavior of these functions and macros:

— `ilogb`, `ilogbf`, `ilogbl`

— `finite`

— `isnan`.

Note

This legacy support will be removed in a future release. ARM recommends that you migrate your use of these features to the equivalent functions in the latest release of the compiler.

- The ARM libraries have been enhanced to provide improved support for multithreading. This is designed to help developers who are working with RTOS-based systems.

  The user-overridable function `__user_libspace()` has been split into two wrapper functions:

  __user_perproc_libspace()

  This returns a pointer to the `__user_libspace` data area used to store data that is global to an entire process, that is, data shared between all threads.

  __user_perthread_libspace()

  This returns a pointer to the `__user_libspace` data area used to store data that is local to a particular thread.

  There are also three new user-overridable functions to manage the locking mechanisms used to prevent corruption of shared data due to concurrent access:

  _mutex_initialize()

  This accepts a pointer to a 32-bit word and initializes it as a valid mutex.

  ```
  int _mutex_initialize(mutex *m);
  ```

  _mutex_acquire()

  This causes the calling thread to obtain a lock on the supplied mutex.

  ```
  void _mutex_acquire(mutex *m);
  ```

_mutex_release()

This causes the calling thread to release the supplied mutex.

```
void _mutex_release(mutex *m);
```

This has resulted in a change in the way some functions behave in a multi threaded environment, making it easier for developers to use these functions in multiprocess systems.

- The ARM librarian, `armar`, supports the command-line option `--diag_style` to specify how diagnostic messages are displayed, for example, to include the line number and character count, use:

  ```
  armar --diag_style ide
  ```

  The default is ARM format, that is, `--diag_style arm`.

2.3.3 Changes to the ARM linker

Be aware of the following differences in the linker in RVCT v2.2:

- The ARM linker conforms to the *Base Platform ABI for the ARM Architecture* [BPABI] and supports the GNU-extended symbol versioning model. The linker offers new options to control symbol versioning, that is, use:
  - `--symver_script` *`file`* to turn on implicit symbol versioning and input *file* as a symbol version script.
  - `--symver_soname` to turn on implicit symbol versioning and version symbols in order to force static binding.

    Where a symbol has no defined version, the linker uses the `SONAME` of the file being linked.

  Note

  In general, symbol versioning is only useful when producing or linking against a DSO or shared library. It has no effect on static linking.

- The ARM linker has global visibility of all your program code and so can now perform some branch optimizations that are not available to other components of RVCT.

  Two new command-line options control these optimizations, that is, use:
  - `--inline` to enable branch inlining. By default, inlining is off.
  - `--info inline` to display information each time a function is inlined and to see the total number of inlines.

  When you specify both `--inline` and `--feedback` *`file`*, functions that are inlined by the linker are also emitted in the feedback file.

- Two new command-line options are available to handle tail calling sections to optimize the branch instruction at the end of the section, that is, use:
  - `--tailreorder` to move tail calling sections above their target, if possible
  - `--info tailreorder` to display information about tail calling sections that have been moved.
- Two new command-line options enable you to control veneer generation in the ARM linker, that is, use:
  - `--no_inlineveneer` to disable inline veneers
  - `--no_veneershare` to prevent veneer sharing.
- The ARM linker supports building, and linking against, shared libraries. New command-line options are available to build SVr4 and BPABI executable files and shared objects, BPABI DLLs, and to specify how code is generated, that is, use:
  - `--sysv` to build an SVr4 formatted ELF file
  - `--shared` to build an SVr4 shared object
  - `--soname` *name* to specify the `SONAME` for a shared object
  - `--fpic` to link position-independent code
  - `--init` *symbol* to specify initialization code
  - `--fini` *symbol* to run code when unloading an executable file or shared object
  - `--linux_abitag` *id* to specify the minimum compatible Linux kernel version
  - `--dynamiclinker` *name* to change the default dynamic linker.
- A new linker option is available, `--startup`, to use alternative C libraries with a different startup symbol. Similarly, the `--cppinit` option is available for C++ initialization code.
- The ARM linker option `--symbols` lists local and global symbols used in the link step. In RVCT v2.2, this output no longer lists mapping symbols by default. A new command-line option, `--list_mapping_symbols`, now enables you to include mapping symbols in the output produced by `--symbols`.
- If the linker detects objects that specify ARMv3 (obsolete in RVCT v2.2), it upgrades these to ARMv4 to be usable with ARM libraries. The linker displays a warning message where it raises the target architecture level.
- The ARM linker enables you to refer to an input section by symbol name in your scatter load description files. See the description of *input_symbol_pattern* in *RealView Compilation Tools v2.2 Linker and Utilities Guide*.

2.3.4 Changes to the ARM assembler

Be aware of the following differences in the assembler in RVCT v2.2:

- ARMv6T2 defines Thumb-2, a major enhancement of the Thumb instruction set. Thumb-2 provides almost exactly the same functionality as the ARM instruction set. It has both 16-bit and 32-bit instructions, and achieves ARM-like performance with Thumb-like code density.

  ARMv6T2 also defines several new instructions in the ARM instruction set.

  The assembler supports all the new instructions in both ARM and Thumb-2.

- RVCT v2.2 includes support for new ARMv6 architecture extensions:
  - ARMv6Z defines ARM Security Extensions (TrustZone), as used in the ARM1176JZ(F)-S core.
  - ARMv6K defines instructions for *Symmetric Multiprocessor* systems (SMP), as used in the ARM MPCore.

- The RVCT v2.2 assembler enables you to write source code that can be assembled to either ARM or Thumb-2 instructions. You can use the same language to write Thumb instructions for pre-Thumb-2 processors.

  However, RVCT v2.2 also supports the old Thumb-2 assembly language to enable the assembly of old code.

- The ARM assembler supports the `COMMON` directive to allocate a block of memory, of the defined size, at the specified symbol. You can also specify how the memory is aligned:

  `COMMON` *`symbol`*`{,`*`size`*`{,`*`alignment`*`}}`

- The ARM assembler provides a new option, `--dwarf3`, to specify DWARF3 standard debug tables. DWARF2 remains the default.

- The ARM assembler offers new options to give symbol visibility when building shared objects or DLLs, that is, use:
  - `--dllexport_all` to provide dynamic visibility of all global symbols unless specified otherwise
  - `--no_hide_all` to export all `extern` definitions and import all undefined references.

- The ARM assembler can output ELF symbols with a visibility set through the use of new attributes to the `IMPORT` and `EXPORT` directives:
  - `DYNAMIC`
  - `HIDDEN`
  - `PROTECTED`

2.3.5 Changes to the fromelf utility

Be aware of the following difference in `fromelf` in RVCT v2.2:

- The fromelf utility has been enhanced to support GNU-extended symbol versioning tables in the output from `--text=/s`. Use the new `--no_symbolversions` option to turn off the decoding of symbol version tables.

Chapter 3
Creating an Application

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

- *Building an application* on page 3-2.

3.1 Building an application

This section describes how to build an application. See:

- *Using the ARM compiler*
- *Using the ARM assembler* on page 3-4
- *Using an IDE* on page 3-6.

3.1.1 Using the ARM compiler

The ARM compiler, `armcc`, can compile C and C++ source into 32-bit ARM code, or 16-bit Thumb code, or 16/32-bit Thumb-2 code.

Typically, you invoke the ARM compiler as follows:

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

You can specify one or more input files.

Building an example

Sample C source code for a range of applications is installed in the main examples directory. 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 the main examples directory, in `...\dhrystone`. This can be used to measure integer processing performance of a system.

To build the Dhrystone example:

1. Compile the C files `dhry_1.c` and `dhry_2.c` with the following options:

   ```
   armcc -c -W -g -O3 -Otime --no_inline --no_multifile -DMSC_CLOCK
   ```

   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 for source-level debugging.
`-O3`	Tells the compiler to generate code with maximum optimizations applied. Used with `-g`, the debug view might be less satisfactory.
`-Otime`	Tells the compiler to optimize the code for speed (not space) for this benchmark.

`--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 (required by Dhrystone).
`-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 v2.2 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 a compatible debugger, for example, RealView Debugger or AXD, to load and run 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 v2.2 Linker and Utilities Guide* for a detailed description of the linker options

- the chapter describing scatter-loading description files in *RealView Compilation Tools v2.2 Linker and Utilities Guide*
- the chapter describing how to develop embedded software in *RealView Compilation Tools v2.2 Developer Guide*.

Using fromelf

An executable image in ELF executable format can be converted to other file formats using the ARM `fromelf` utility. For more information, see the chapter describing `fromelf` in *RealView Compilation Tools v2.2 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` that contains a detailed listing of the assembly language produced by the assembler.

For full details on the options and syntax, see *RealView Compilation Tools v2.2 Assembler Guide*.

Building an example

Example 3-1 on page 3-5 shows a small ARM/Thumb interworking assembly language program. You can use it to explore the use of the assembler and linker.

To build the example:

1. Copy the code from the PDF or DynaText online documentation and save it in a 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 a 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 v2.2 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.
```

3.1.3 Using an IDE

An *Integrated Development Environment* (IDE) enables you to use a graphical interface to manage your software development projects. An IDE lets you configure the ARM tools to compile, assemble, and link your project code.

You can invoke the ARM tools to develop C, C++, and assembly language programs through an IDE, for example, if you are using RealView Debugger or ADS.

ARM has also licensed the CodeWarrior IDE from Metrowerks and is making this available with RealView Developer Suite. However, this is not available on Sun Solaris or Red Hat Linux platforms.

See *RealView Developer Suite Getting Started Guide* for more details on using these IDEs.

Appendix A
About Previous Releases

This appendix describes the major differences between previous releases of *RealView® Compilation Tools*, that is, RVCT v2.1, RVCT v2.0, and RVCT v1.2.

It contains the following sections:

A.1 Differences between RVCT v2.1 and RVCT v2.0

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

- *General changes*
- *Changes to the ARM compiler* on page A-3
- *Changes to library support* on page A-4
- *Changes to the ARM linker* on page A-4
- *Changes to the ARM assembler* on page A-5
- *Changes to the fromelf utility* on page A-5.

A.1.1 General changes

The following changes were made in RVCT v2.1:

- Full support for C++, including exceptions.
- Enhanced support for ARMv6 cores. To see a full list of supported cores use:

  ```
  armcc --cpu list
  ```

- The compiler, linker, assembler, and `fromelf` 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.
- 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.
- The ARM® tools can share suitable strings across compile units using `SHF_STRINGS` sections, as defined by the *ELF for the ARM Architecture* 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 softfpa`).

In RVCT v2.1, 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.

A.1.2 Changes to the ARM compiler

The following changes were made in 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*.

- 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.

- 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, that includes multifile compilation by default.

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

- New `__breakpoint()` intrinsic.

- Noreturn functions.

A.1.3 Changes to library support

The following changes were made in 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 `wchar.h` function, except file I/O, 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`.

A.1.4 Changes to the ARM linker

The following changes were made in RVCT v2.1:

- 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`*

  For full details on this change, see *Compatibility with legacy objects and libraries* on page A-9.

A.1.5 Changes to the ARM assembler

The following changes were made in RVCT v2.1:

- New `--cpu list` and `--fpu list` options to display details about CPU and architectures supported.
- The assembler includes the new `:RCONST:` unary operator to return the number of a given register.
- 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*).
- The assembler can give warnings about possible interlocks in your code. To enable this, use:

  ```
  armasm --diag_warning 1563
  ```

A.1.6 Changes to the fromelf utility

The following change was made in 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.

A.2 Differences between RVCT v2.0 and RVCT v1.2

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

- *General changes*
- *Changes to the ARM compiler*
- *Changes to the ARM linker* on page A-7
- *Changes to the ARM assembler* on page A-8.

A.2.1 General changes

The following changes were made in 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 `--fpmode ieee_full`. This is because the default setting is now `--fpmode std` and so floating-point exceptions are not generated by default.
- Support for double dashes "--" to indicate command-line keywords (for example, `--cpp`).

A.2.2 Changes to the ARM compiler

The following changes were made to the ARM compiler (`armcc`):

- 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 `--fpmode` option.

- The behavior of the `--list` option is different from that in the older compilers.
- 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 `#nnnn-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 v2.0 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 v2.0 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.

A.2.3 Changes to the ARM linker

The following changes were 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.

A.2.4 Changes to the ARM assembler

The following changes were 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}`.

A.3 Compatibility with legacy objects and libraries

The rest of this appendix describes the level of compatibility provided and the restrictions:

- *Compatibility of RVCT v2.2 with legacy RVCT v2.1/RVCT v2.0 objects and libraries*.
- *Compatibility of RVCT v2.2 with legacy RVCT v1.2/ADS v1.2 objects and libraries* on page A-10.

A.3.1 Compatibility of RVCT v2.2 with legacy RVCT v2.1/RVCT v2.0 objects and libraries

Backward compatibility of object/library code is supported (for example, legacy RVCT v2.0- or v2.1-built objects are compatible with RVCT v2.2), as long as you use the RVCT v2.2 linker and C/C++ libraries. Forward compatibility is not guaranteed.

For example, legacy RVCT v2.1 objects will normally be linkable with RVCT v2.2 objects, using the RVCT v2.2 linker and RVCT v2.2 C/C++ libraries.

An unlikely exception to this would be if you had compiled your RVCT v2.1 objects with a feature that was deprecated in RVCT v2.1 (for example, `--fpu SoftFPA`) and subsequently removed in RVCT v2.2.

You must link using the RVCT v2.2 linker, not the linker of older ARM tools. This is because older linkers cannot process objects produced by the RVCT v2.2 compilers.

You must link with the RVCT v2.2 C and C++ runtime libraries supplied by ARM, because these provide all the necessary support functions required by RVCT v2.2-compiled code.

A.3.2 Compatibility of RVCT v2.2 with legacy RVCT v1.2/ADS v1.2 objects and libraries

The ABI in RVCT v2.x is different to that in ADS v1.2 and RVCT v1.2. Therefore, legacy ADS v1.2 and RVCT v1.2 objects and libraries are not directly compatible with RVCT v2.x. Some restricted compatibility is provided with the `--apcs /adsabi` compiler option. However this option is deprecated and will be removed in a future release. For more details, see the chapter on using the ARM compiler in *RealView Compilation Tools v2.2 Compiler and Libraries Guide*.

When linking with legacy RVCT v1.2/ADS v1.2 objects/libraries, be aware of the following:

- To enable RVCT v2.2 C objects to be used with legacy RVCT v1.2/ADS v1.2 C or C++ objects/libraries, compile the RVCT v2.2 C code with `--apcs /adsabi`.
- To enable RVCT v2.2 C++ objects to be used with legacy C RVCT v1.2/ADS v1.2 objects/libraries, compile the RVCTv2.2 C++ code with `--apcs /adsabi`.

  Note

  This can only work if your C++ code does *not* use any of the Rogue Wave C++ libraries, because these libraries are incompatible with objects compiled with `--apcs /adsabi`.

- There is no compatibility possible between RVCT v2.x C++ objects/libraries and legacy RVCT v1.2/ADS v1.2 C++ objects/libraries.
- You must link using the RVCT v2.2 linker, not the ADS v1.2 linker, because the ADS v1.2 linker cannot process objects compiled with `--apcs /adsabi`.
- You must link with the RVCT v2.2 ARM-supplied C and C++ runtime libraries, because these have been built in a special way to be compatible with objects compiled with `--apcs /adsabi`.

Given these restrictions, ARM *strongly* recommends that you rebuild your *entire* project, including any user-supplied libraries, with RVCT v2.2 to avoid any potential incompatibilities, and to take full advantage of the improved optimization, enhancements, and new features provided by RVCT v2.2.

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.

ARM Developer Suite (ADS)
A suite of software development applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of *RISC* processors. ADS is superseded by RealView Developer Suite (RVDS).

See also RealView Developer Suite.

ARM instruction A word that encodes an operation for an ARM processor operating in ARM state. ARM instructions must be word-aligned.

ARM state A processor that is executing ARM instructions is operating in ARM state. The processor switches to Thumb state (and to recognizing Thumb instructions) when directed to do so by a state-changing instruction such as `BX`, `BLX`.

See also Thumb state.

armar The ARM librarian.

armasm The ARM assembler.

armcc The ARM compiler.

armlink The ARM linker.

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.

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 by default. From RVCT v2.2, you can optionally generate DWARF3 format (Draft Standard 9).

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.

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 An object (variable or function) within an image that has global scope.

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

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.

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.

Local An object that is only accessible to the subroutine that created it.

MPCore An integrated Symmetric Multiprocessor system (SMP) delivered as a traditional uniprocessor core. The chip contains up to four ARM1136J-S based CPUs with cache coherency.

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.

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 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.

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 latest version of the ARM simulator, RealView ARMulator ISS is supplied with RealView Developer Suite.

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.

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 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.

Reduced Instruction Set Computing (RISC)
Device where the number of instructions a microprocessor runs for a specific application are reduced from a general purpose Complex Instruction Set Computing (CISC) device to create a more efficient operating system.

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

Register A processor register.

RISC *See* Reduced Instruction Set Computing.

ROM Read Only Memory.

ROPI *See* Read-Only Position Independent.

RVCT *See* RealView Compilation Tools.

RVDS *See* RealView Developer Suite.

RVISS *See* RealView ARMulator ISS.

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.

SPSR Saved Program Status Register.

See also Program Status Register.

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.

Stack Pointer (SP) Integer register R13.

SVr4 *See* System V release 4.

SWI *See* Software Interrupt.

System V release 4 (SVr4)
The System V Application Binary Interface (ABI) is a family of specifications that define a standard binary interface for compiled applications that implement UNIX System V release 4.

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 One halfword or two halfwords that encode an operation for an ARM processor operating in Thumb state. Thumb instructions must be halfword-aligned.

Thumb state A processor that is executing Thumb instructions is operating in Thumb state. The processor switches to ARM state (and to recognizing ARM instructions) when directed to do so by a state-changing instruction such as `BX`, `BLX`.

See also ARM state.

TrustZone ARM technology-optimized software that provides a secure execution environment to enable trusted programs and data to be separated from the operating system and applications.

Unsigned data types
Represent a non-negative integer in the range 0 to + 2N-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

D

E

F

I

L

M

N

O

P

R

S

T

