RealView™ Compilation Tools

Version 2.0

Essentials Guide

ARM®

ARM DUI 0202C

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 2.0

Proprietary Notice

Confidentiality Status

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

Product Status

The information in this document is final (information on a developed product).

Web Address

`http://www.arm.com`

Contents
RealView Compilation Tools Essentials Guide

Preface

This preface introduces the *RealView™ Compilation Tools v2.0 Essentials Guide* and other user documentation. It contains the following sections:

About this book

This book provides an overview of the *RealView Compilation Tools* (RVCT) v2.0 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 v2.0, RVCT v1.2, and the *ARM Developer Suite v1.2* (ADS v1.2).

Chapter 3 ***Creating an Application***

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

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 v2.0 Developer Guide* (ARM DUI 0203). This book provides tutorial information on writing code targeted at the ARM family of processors.
- *RealView Compilation Tools v2.0 Assembler Guide* (ARM DUI 0204). This book provides reference and tutorial information on the ARM assembler.
- *RealView Compilation Tools v2.0 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 v2.0 Linker and Utilities Guide* (ARM DUI 0206). This book provides reference information on the command-line options to the linker and the fromELF utility.

The following additional documentation is provided with RealView Compilation Tools:

- *ARM FLEXlm License Management Guide* (ARM DUI 0209). This is supplied in DynaText format as part of the online books, and as a PDF file in *install_directory*`\Documentation\FLEXlm\3.0\release\`*platform*`\PDF`.
- *ARM ELF specification* (SWS ESPC 0003). This is supplied as a PDF file, `ARMELF.pdf`, in *install_directory*`\Documentation\Specifications\1.0\release\`*platform*`\PDF`.
- *TIS DWARF 2 specification*. This is supplied as a PDF file, `TIS-DWARF2.pdf`, in *install_directory*`\Documentation\Specifications\1.0\release\`*platform*`\PDF`.

- *ARM-Thumb Procedure Call Standard specification*. This is supplied as a PDF file, `ATPCS.pdf`, in *`install_directory`*`\Documentation\Specifications\1.0\release\`*`platform`*`\PDF`.

In addition, refer to the following documentation for specific information relating to ARM products:

- *RealView ARMulator ISS v1.3 User Guide* (ARM IDE 0170)
- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- the ARM datasheet or technical reference manual for your hardware device.

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 RealView Compilation Tools, 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 v2.0* (RVCT v2.0) and describes its software components and documentation. It contains the following sections:

1.1 About the 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*
- *Utilities* 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. The compiler is tested against the Plum Hall C Validation Suite for ISO conformance. It compiles:

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

armasm 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 online HTML documentation on the CD ROM.

support libraries

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

Utilities

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

fromELF The ARM image conversion utility. This accepts 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 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 the compiler, linker, and ELF/DWARF2 compatible debuggers.

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

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

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

EABI The ARM *Embedded Application Binary Interface* (EABI) enables you to use the ARM and Thumb objects and libraries from different producers that support the EABI For more details, see the `ARM EABI home page` on the ARM DevZone™. You can access the ARM DevZone from `http://www.arm.com`.

Supporting software

To debug your programs under simulation, or on ARM-based hardware, use an ELF/DWARF2 compatible debugger.

The following support software is available to enable you to debug your programs under simulation:

RealView ARMulator® ISS

RealView ARMulator Instruction Set Simulator (RealView ARMulator ISS) is the ARM core simulator. This provides instruction-accurate simulation of ARM processors, and enables ARM and Thumb executable programs to be run on non-native hardware. RealView ARMulator ISS provides a series of modules that:

- model the ARM processor core
- model 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 the *RealView ARMulator ISS v1.3 User Guide*.

1.1.2 Documentation

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

1.2 Online documentation

The RVCT documentation is also available online as DynaText electronic books. The content of the DynaText manuals is identical to that of the PDF documentation.

In addition, documentation for the Rogue Wave C++ library is available in HTML format. See *HTML* on page 1-12 for more information.

PDFs of the RVCT manuals are installed only for a Full installation. The Typical installation only installs PDFs of related documentation that is not available in the DynaText online books.

1.2.1 DynaText

The manuals for RVCT are provided on the CD-ROM as DynaText electronic books. The DynaText browser is installed by default for a Typical or Full installation.

To display the online documentation, either:

- select **Online Books** from the **ARM RealView Compilation Tools v2.0** program group
- execute `Dtext.exe` in *`install_directory`*`\Documentation\DynaText\4.1\release\win_32-pentium\bin`.

The DynaText browser displays a list of available collections and books (Figure 1-1 on page 1-6).

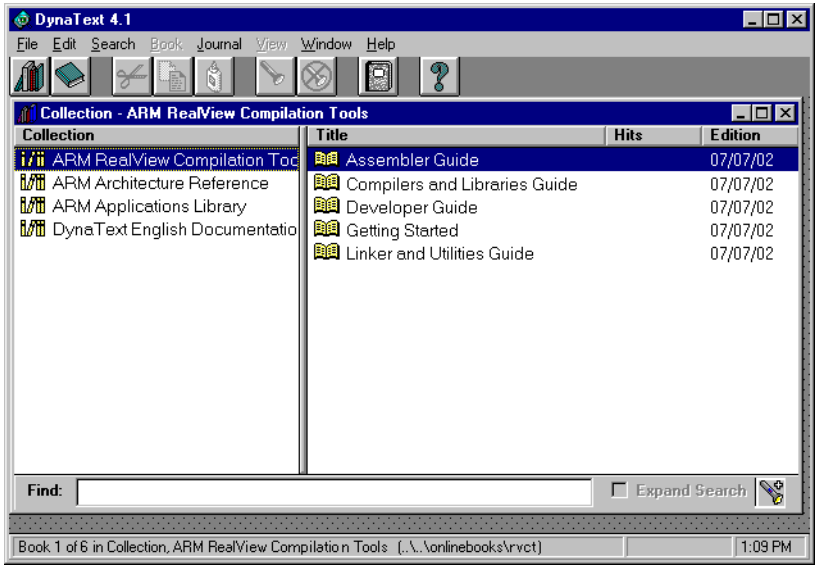

Figure 1-1 DynaText browser with list of available books

Opening a book

Double-click on a title in the book list to open the book. The table of contents for the book is displayed in the left panel and the text is displayed in the right panel (see Figure 1-2 on page 1-7).

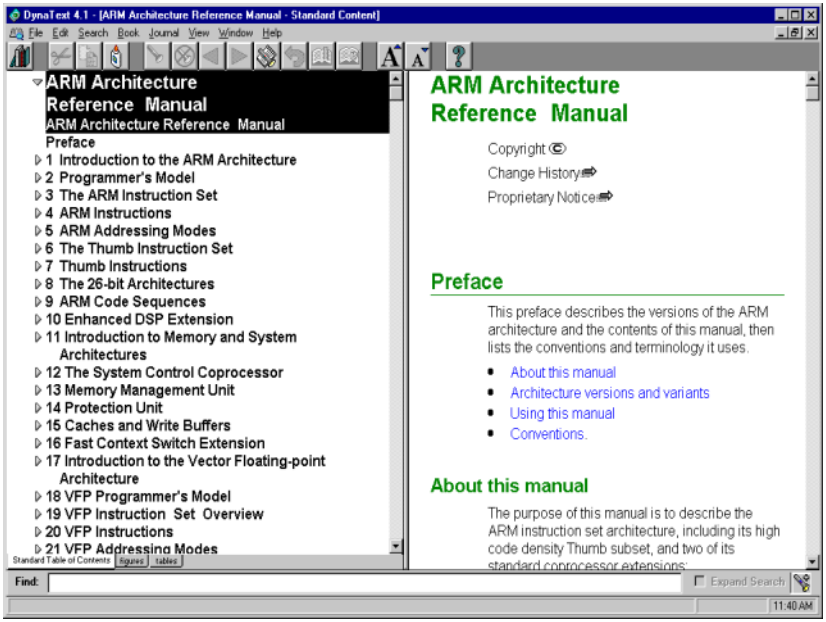

Figure 1-2 Opening a book

Navigating through the book

Click on a section in the table of contents to display the text for that section. For example, selecting *C and C++ libraries* displays the text for that section (see Figure 1-3 on page 1-8).

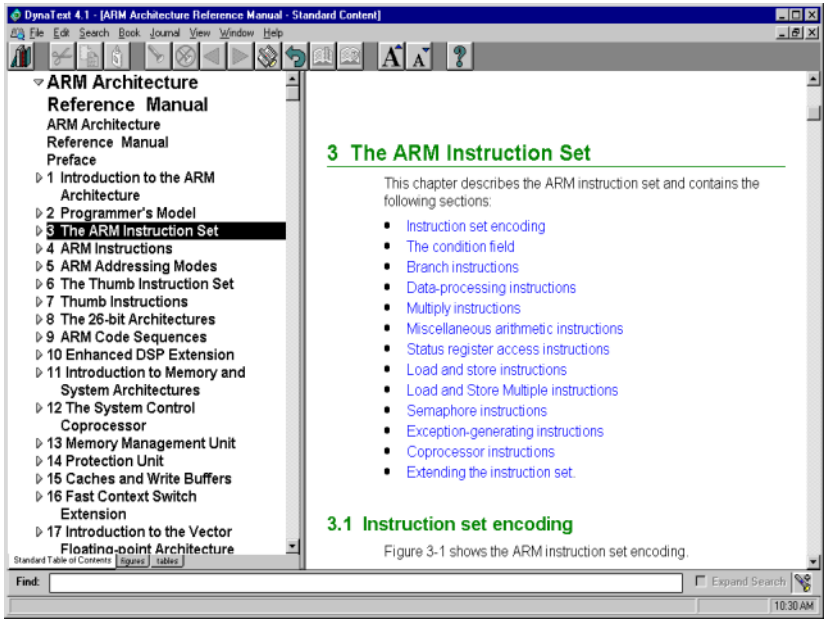

Figure 1-3 Selecting a section from the table of contents

Navigating using hyperlinks

Text in blue indicates a link that displays a different section of a book, or a different book. Plain blue text indicates that the link is within the current chapter. Underlined blue text indicates that the link is either to another chapter within the current book, or to a different book. Hyperlinks behave differently depending on their target:

- if the link is within the current chapter (plain blue text), DynaText scrolls the current window to display the target
- if the link is to another chapter in the current book, DynaText opens a new window without a Table of Contents
- if the link is to another book, DynaText opens a new window with a Table of Contents.

Figure 1-4 on page 1-9 shows the browser displaying the text for the linked text.

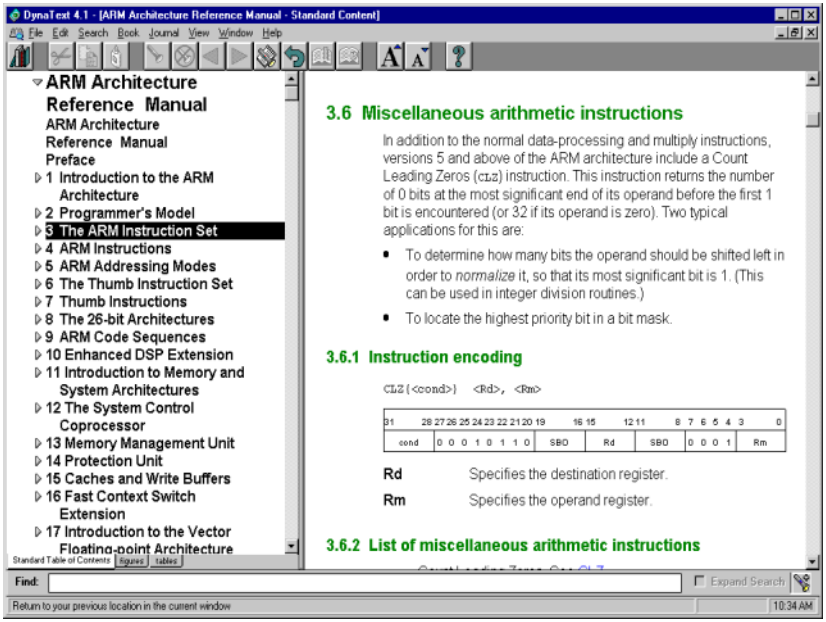

Figure 1-4 Using text links

Displaying graphics

Graphics are not displayed inline in the DynaText browser. If a graphic symbol is displayed, select it to display the linked graphic in its own window (see Figure 1-5).

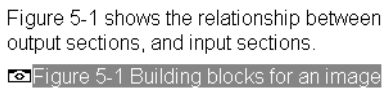

Figure 1-5 Link to a figure

Clicking on the figure icon displays the figure in its own window (see Figure 1-6 on page 1-10).

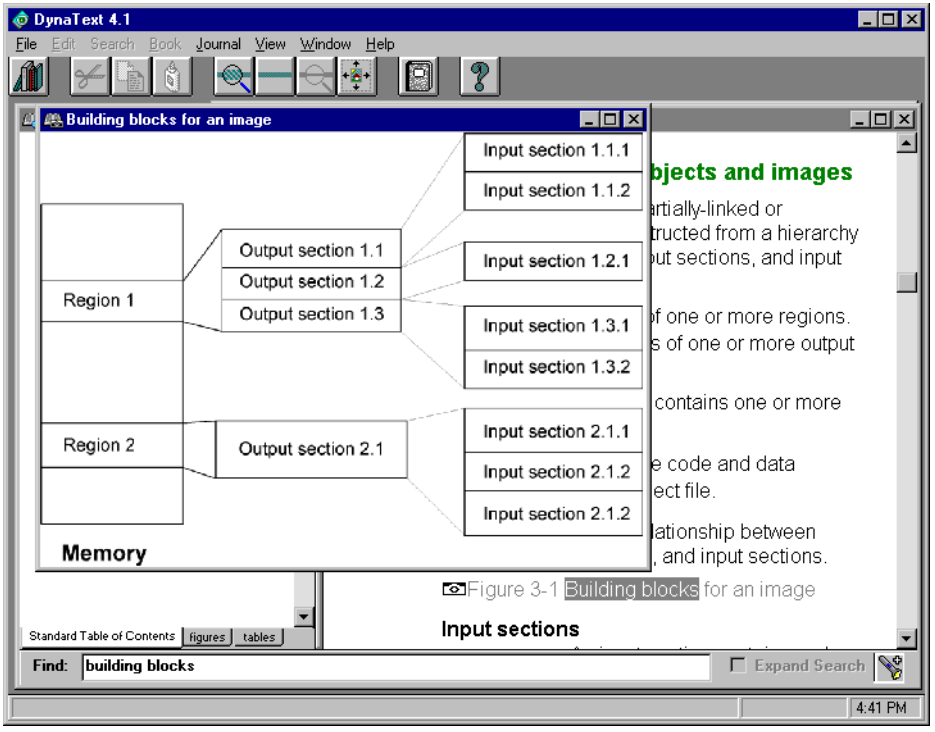

Figure 1-6 Graphic displayed

Navigating to a different book

If the blue link text refers to a different book, clicking on the link text displays the linked book in its own window (see Figure 1-7 on page 1-11).

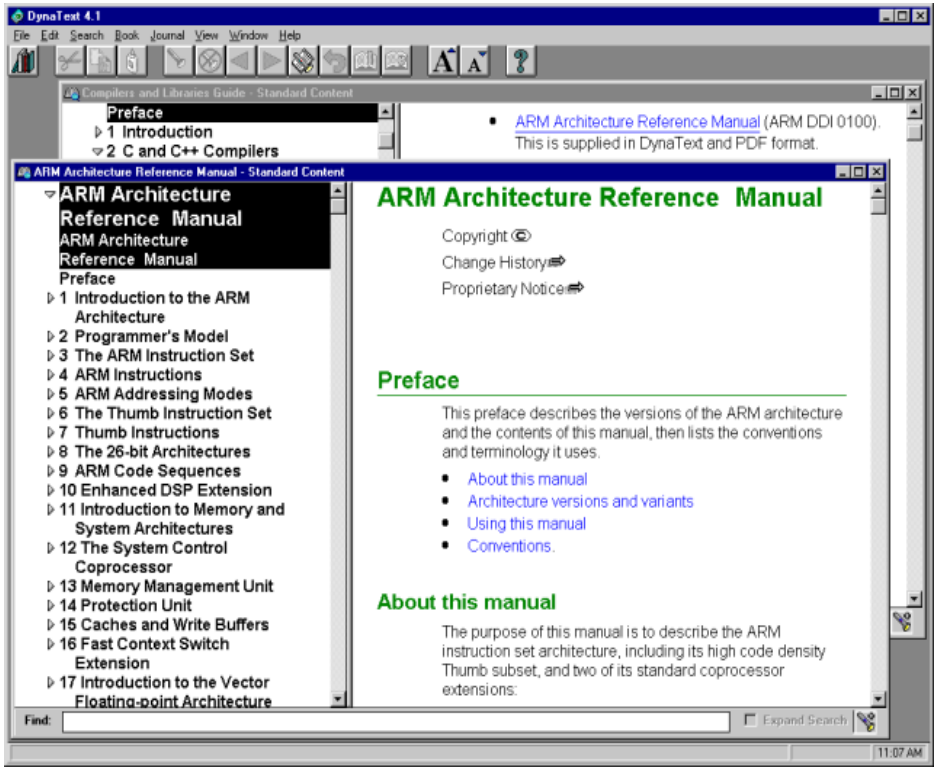

Figure 1-7 Navigating to a different book

Displaying help for DynaText

Select **Help** → **Reader Guide** to display help on how to use DynaText.

1.2.2 HTML

The manuals for the Rogue Wave C++ library for RVCT are provided on the CD-ROM in HTML files. Use a web browser, such as Netscape Communicator or Internet Explorer, to view these files. For example, select *install_directory*\Documentation\RogueWave\1.0\release\stdref\index.htm to display the HTML documentation for Rogue Wave (see Figure 1-8 where the *install_directory* is C:\Program Files\ARM).

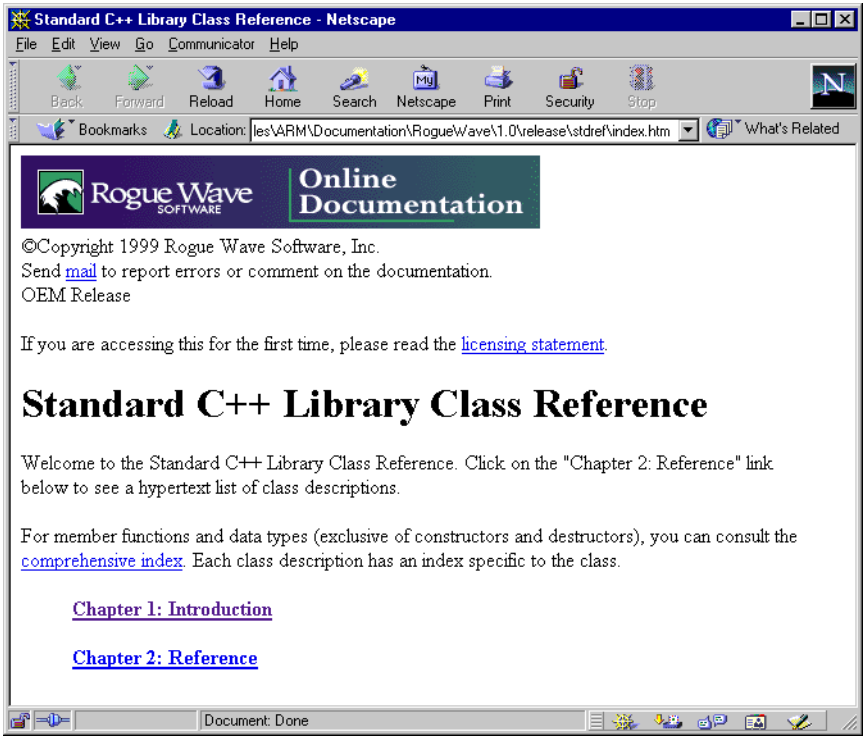

Figure 1-8 HTML browser

Chapter 2
Differences

This chapter describes the major differences between RVCT v2.0, RVCT v1.2, and ADS v1.2. It contains the following sections:

2.1 Overview

This chapter describes the changes that have been made between RVCT v2.0, RVCT v1.2, ADS v1.2.

The most important differences between RVCT v2.0 and RVCT v1.2 are:

- The RVCT v2.0 compiler:
  - 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 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.
  - The RVCT v2.0 compiler is compliant with the ARM *Embedded Application Binary Interface* (EABI).
- There is full ISO C++ support as defined by the *ISO/IEC 14822 :1998 International Standard for C++*, by way of the EDG (Edison Design Group) front-end. This includes namespaces, templates and intelligent implementation of *Run-Time Type Information* (RTTI), but excludes exceptions.
- Support for ARM Architecture v6.
- Compiance with the ARM *Embedded Application Binary Interface* (EABI).
- Re-engineered inline assembler and new embedded assembler.
- ARM and Thumb compilation on a per-function basis.
- Unicode and multibyte characters are supported.
- There are no temporary licenses.

For a summary of the changes between RVCT v1.2 and ADS v1.2, see *Changes between RVCT v1.2 and ADS v1.2* on page 2-6.

2.2 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 2-4
- *Changes to the ARM assembler* on page 2-5.

2.2.1 General changes

The following changes have been made to RVCT:

- Support for ARM architecture v6.
- Compliance with the ARM *Embedded Application Binary Interface* (EABI). See the `ARM EABI home page` on the ARM DevZone™. You can access the ARM DevZone from `http://www.arm.com/`.
- Support for double dashes "--" to indicate command-line keywords (for example, `--cpp`) and single dashes "-" for command-line single-letter options, with or without arguments (for example, `-S`).

  Note

  The single-dash command-line options used in previous versions of ADS and RVCT are still supported for backwards-compatibility.

2.2.2 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, `#pragma arm` and `#pragma thumb`.

- Four 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 the *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 the *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.

2.2.3 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.

2.2.4 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}.

2.3 Changes between RVCT v1.2 and ADS v1.2

The most important differences between RVCT v1.2 and ADS v1.2 are:

- The CodeWarrior IDE for managing projects has been removed.
- The *ARM eXtended Debugger* (AXD) and *ARM symbolic debugger* (`armsd`) have been removed.
- The RealView ARMulator instruction set simulator is not included in RVCT. This is now provided with other ARM debuggers and as a separate product.
- The *ARM Profiler* (`armprof`) has been removed.
- The ARM Applications Library is not included.

Chapter 3
Creating an Application

This chapter describes how to create an application using RVCT. It contains the following sections:

3.1 Building an application

This section describes how to build an application using:

- the compiler (see *Using the compilers*)
- the assembler (see *Using the assembler* on page 3-5)
- the linker (see *Setting linker options* on page 3-6).

3.1.1 Using the compilers

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

Typically, the ARM compiler is invoked as follows:

`armcc [options] ifile_1 ... ifile_n`

You can specify one or more input files *`ifile_1 ... ifile_n`*. 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 start-up configuration is adjusted by the filename extension you specify.

Table 3-1 Start-up configuration as adjusted by filename extension

Filename extension	Instruction set	Source language
.cpp	No adjustment	C++
.c	No adjustment	No adjustment
.tc	Thumb	C
.tcpp	Thumb	C++
.ac	ARM	C
.acpp	ARM	C++

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 v2.0. The startup configuration associated with each of the older tool names is shown in Table 3-2.

Table 3-2 Start-up configuration based on old tool names

Tool name	Instruction set	Source language
armcc[a]	ARM	C
tcc	Thumb	C
armcpp	ARM	C++
tcpp	Thumb	C++

a. This is included for completeness, even though it is the same tool name as the ARM compiler for RVCT v2.0.

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 Start-up configuration as adjusted by overriding options

Command-line option	Instruction set	Source language
--c90	No adjustment	C
--cpp	No adjustment	C++
--arm	ARM	no adjustment
--thumb	Thumb	no adjustment

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 3-4 Example configuration

Example Command Component	Description	Instruction set	Source language
`tcpp`	tool name	Thumb	C++
`.acpp`	filename extension	ARM	C++
`--c90`	command-line option	No adjustment	C
	Result	ARM	C

To summarize, the filename extension overrides the default configuration determined by the tool name used 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 simple application is in *install_directory*\Rvct\Examples\2.0*build_num*\windows\embedded\embed\main.c.

To build the example:

1. Compile the C file `main.c` with either:

   `armcc -g -O1 -c main.c` (for ARM)

   `armcc --thumb -g -O1 -c main.c` (for Thumb)

   where:

   `-g` Tells the compiler to add debug tables.

   `-O1` Tells the compiler to select the best possible optimization while maintaining an adequate debug view.

   `-c` Tells the compiler to compile only (not to link).

   `--thumb` Tells the compiler to generate Thumb code. (The alternative option, `--arm`, tells the compiler to generate ARM code, and is the default.)

2. Link the image using the following command:

   ```
   armlink main.o -o embed.axf
   ```

   where:

   `-o` Specifies the output file as `embed.axf`.

3. Use an ELF/DWARF2 compatible debugger to load and test the image.

3.1.2 Using the 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, refer to the *RealView Compilation Tools v2.0 Assembler Guide*.

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

Example 3-1

```
     AREA     AddReg,CODE,READONLY        ; Name this block of code.
     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.
```

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. Type `armasm -list addreg.lst addreg.s` at the command prompt to assemble the source file.
3. Type `armlink addreg.o -o addreg` to link the file.
4. Use an ELF/DWARF2 compatible debugger 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 the *RealView Compilation Tools v2.0 Assembler Guide*.

3.1.3 Setting linker options

The ARM linker, `armlink`, enables you to:

- link a collection of objects and libraries into an executable ELF image
- partially link a collection of objects into an object that can be used as input for a future link step
- specify where the code and data will be located in memory
- produce debug and reference information about the linked files.

Objects consist of input sections that contain code, initialized data, or the locations of memory that must be set to zero. Input sections can be *Read-Only* (RO), *Read/Write* (RW), or *Zero-Initialized* (ZI). These attributes are used by `armlink` to group input sections into bigger building blocks called output sections, regions and images. Output sections are approximately equivalent to ELF segments.

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-load description file.

Linker input and output

Input to armlink consists of:

- one or more object files in ELF Object Format
- optionally, one or more libraries created by armar.

Output from a successful invocation of armlink is one of the following:

- an executable image in ELF executable format
- a partially linked object in ELF object format.

For simple images, ELF executable files contain segments that are approximately equivalent to RO and RW output sections in the image. An ELF executable file also has ELF sections that contain the image output sections.

An executable image in ELF executable format can be converted to other file formats by using the fromELF utility. See the *RealView Compilation Tools v2.0 Linker and Utilities Guide* for information on the fromELF utility.

Linker syntax

The linker command syntax is of the form:

armlink [*-help_options*] [*-output_options*] [*-via_options*] [*-memory_map_options*] [*-image_content_options*] [*-image_info_options*] [*-diagnostic_options*]

See the *RealView Compilation Tools v2.0 Linker and Utilities Guide* for a detailed list of the linker options.

Using linker options to position sections

The following linker options control how sections are arranged in the final image and whether the code and data can be moved to a new location after the application starts:

-ropi
: This option makes the load and execution region containing the RO output section position-independent. If this option is not used the region is marked as absolute.

-ro-base *address*
: This option sets the execution addresses of the region containing the RO output section at *address*. The default address is 0x8000.

-rw-base *address*
: This option sets the execution addresses of the region containing the RW output section at *address*. The default address is at the end of the RW section.

`-rwpi` This option makes the load and execution region containing the RW and ZI output section position-independent. If this option is not used the region is marked as absolute. The `-rwpi` option is ignored if `-rw-base` is not also used. Usually each writable input section must be read-write position-independent.

If you want more control over how the sections are placed in an image, use the `-scatter` option and specify a scatter-load description file.

Using scatter-load description files for a simple image

The command-line options (`-ro-base`, `-rw-base`, `-ropi`, and `-rwpi`) create simple images.

You can create the more complex images by using the `-scatter` option to specify a scatter-load description file. The `-scatter` option is mutually exclusive with the use of any of the simple memory map options `-ro-base`, `-rw-base`, `-ropi`, or `-rwpi`.

For more information on the linker and scatter-load description files, see the *RealView Compilation Tools v2.0 Linker and Utilities Guide* and the chapter on embedded software development in the *RealView Compilation Tools v2.0 Developer Guide*.

3.2 Using ARM libraries

The following run-time libraries are provided to support compiled C and C++:

ISO C The C libraries consist of:

- The functions defined by the ISO C library standard.
- Target-dependent functions used to implement the C library functions in the semihosted execution environment. You can redefine these functions in your own application.
- Helper functions used by the C and C++ compilers.

C++ The C++ libraries contain the functions defined by the ISO C++ library standard. The C++ library depends on the C library for target-specific support and there are no target dependencies in the C++ library. This library consists of:

- the Rogue Wave Standard C++ Library version 2.01.01
- helper functions for the C++ compiler
- additional C++ functions not supported by the Rogue Wave library.

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 the ARMulator, RealMonitor, Angel, and Multi-ICE. You can use the ARM development tools in RVCT to develop applications. To load and test the application under the ARMulator or on a development board, use an ELF/DWARF2 compatible debugger. See the chapter on Semihosting in the *RealView Compilation Tools v2.0 Compiler and Libraries Guide* for more information.

You can re-implement any of the target-dependent functions of the C library as part of your application. This enables you to tailor the C library, and therefore the C++ library, to your own execution environment.

The libraries are installed in the following subdirectories within *install_directory*\RVCT\Data\2.0*build_num*\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\2.0*build_num*\include.

`cpplib` Contains the variants of the Rogue Wave C++ library and supporting C++ functions. The Rogue Wave and supporting C++ functions are collectively referred to as the ARM C++ Libraries. The accompanying header files are installed in *install_directory*\RVCT\Data\2.0*build_num*\include.

Note

- The ARM C libraries are supplied in binary form only.
- The ARM libraries should 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, and include it when you link the application. Your version of the function will be used instead of the standard library version.
- Normally, only a few functions in the ISO C library require re-implementation in order to create a target-dependent application.
- The source for the Rogue Wave Standard C++ Library is not freely distributable. It can be obtained from Rogue Wave Software Inc., or through ARM Limited, for an additional licence fee. See the Rogue Wave online documentation in *install_directory*\Documentation\RogueWave\1.0\release for more about the C++ library.

3.2.1 Using the ARM libraries in a semihosted environment

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, ARMulator, RealMonitor, Angel, and Multi-ICE
- 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 library.

See the chapter on Semihosting in the *RealView Compilation Tools v2.0 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.
2. Use `#pragma import(__use_no_semihosting_swi)` 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. See the chapter on C and C++ libraries in the *RealView Compilation Tools v2.0 Compiler and Libraries Guide* for more information.

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`
- 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 run-time 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. See the chapter on C and C++ libraries in the *RealView Compilation Tools v2.0 Compiler and Libraries Guide* for more information.

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 the 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 the 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 *RealView Compilation Tools v2.0 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

American National Standards Institute (ANSI)
: An organization that specifies standards for, among other things, computer software. This is superseded by the International Standards Organization.

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

ANSI
: *See* American National Standards Institute.

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.

Archive
: A package containing all the files associated with a release of a built model.

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.

armasm
: The ARM assembler.

armcc The ARM C compiler.

ARM-Thumb Procedure Call Standard (ATPCS)
Defines how registers and the stack are used for subroutine calls.

ARMulator *See* RealView ARMulator ISS.

ATPCS *See* ARM-Thumb Procedure Call Standard.

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

Bit Binary digit.

Breakpoint A location in the image. If execution reaches this location, the debugger halts execution of the image.

See also Watchpoint.

Byte An 8-bit data item.

C file A file containing C source code.

Cache A block of high-speed memory locations whose addresses are changed automatically in response to those memory locations the processor is accessing, and whose purpose is to increase the average speed of a memory access.

Class A C++ class involved in the image.

CLI C Language Interface/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 ADW, ADU and AXD all 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.

CPSR Current Program Status Register.

See also Program Status Register.

CPU Central Processor Unit.

C, C++ Programming languages.

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

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 Debug With Arbitrary Record Format.

Dynamic Linked Library
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 Executable and Linking Format.

Embedded 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.

FIQ Fast Interrupt.

Flash memory Nonvolatile memory that is often used to hold application code.

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

FPA Floating-Point Accelerator.

FPE Floating-Point Emulator.

FPU Floating-Point Unit.

GCC The Gnu C Compiler.

Global variables Variables with global scope within the image.

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.

ICE In-Circuit Emulator.

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 allow small numeric values to be encoded as immediate values within the instruction that operates on them.

Inline Functions that are repeated in code each time they are used rather than having a common subroutine. Assembler code placed within a C or C++ program.

See also Embedded.

Input section Contains code or initialized data or describes a fragment of memory that must be set to zero before the application starts.

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

IRQ Interrupt Request.

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

ISO *See* International Standards Organization.

I/O In/out.

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)
Allows 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 multi-processor JTAG-based debug tool for embedded systems. Multi-ICE is an ARM registered trademark.

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.

Processor An actual processor, real or emulated running on the target. A processor always has at least one context of execution.

Processor Status Register
See Program Status Register.

Program Counter (PC)
Integer register R15.

Program Status Register (PSR)
Contains some information about the current program and some information about the current processor. Often, therefore, also referred to as Processor Status Register. 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
Code and read-only data addresses can be changed at run-time.

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

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

RealView ARMulator ISS
RealView ARMulator instruction set simulator (RealView ARMulator ISS). A collection of modules that simulate the instruction sets and architecture of various ARM processors. See the *RealView ARMulator ISS v1.3 User Guide* for more information.

Register A processor register.

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

- a debug agent running on the host (for example, ARMulator)
- a debug monitor running on ARM-based hardware accessed through a communication link (for example, Angel)
- a debug agent controlling an ARM processor through hardware debug support (for example, 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.

RPS Reference Peripheral System.

RWPI *See* Read Write Position Independent.

Saved Program Status Register
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.

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
: Software Development Toolkit (SDT) is an ARM product still supported but superseded by the ARM Development Suite (ADS) and RealView Compilation Tools (RVCT).

Source File
: A file that is processed as part of the image building process. Source files are associated with images.

SP (Stack Pointer)
: Integer register R13.

SPSR
: Saved Program Status Register.

  See also Program Status Register.

Software Interrupt (SWI)
: An instruction that causes the processor to call a programer-specified subroutine. Used by ARM to handle semihosting.

SWI
: *See* Software Interrupt.

Target
: The actual target processor, (real or simulated), on which the application is running.

  The fundamental object in any debugging session. The basis of the debugging system. The environment in which the target software will run. It is essentially a collection of real or simulated processors.

TCC
: Thumb C Compiler.

Thumb instruction
: A halfword that specifies an operation for an ARM processor in Thumb state to perform. Thumb instructions must be halfword-aligned.

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 + 2N-1, using normal binary format.

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

VFP
: 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 otherwised 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.

O

R

S

