RealView® Developer Kit

Version 2.2

Compiler and Libraries Guide

ARM®

RealView Developer Kit
Compiler and Libraries Guide

Copyright © 2005 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
April 2005	B	Release for RVDK v2.2

Proprietary Notice

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

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

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

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

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

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

Web Address

`http://www.arm.com`

Contents
RealView Developer Kit Compiler and Libraries Guide

Preface

This preface introduces the *RealView® Developer Kit (RVDK) v2.2 Compiler and Libraries Guide*. It contains the following sections:

About this book

This book provides reference information for *RealView® Developer Kit* (RVDK), and describes the command-line options to the ARM® compiler. The book also gives reference material on the ARM implementation of C and C++ in the compiler and the C and C++ libraries.

Intended audience

This book is written for all developers who are producing applications using RVDK. 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 the ARM compiler for RVDK and the libraries.

Chapter 2 *Using the ARM Compiler*

> Read this chapter for an explanation of all command-line options accepted by the ARM compiler.

Chapter 3 *ARM Compiler Reference*

> Read this chapter for a description of the language features provided by the ARM compiler, and for information on standards conformance and implementation details.

Chapter 4 *Inline and Embedded Assemblers*

> Read this chapter for a description of the inline and embedded assemblers provided by the ARM compiler.

Chapter 5 *The C and C++ Libraries*

> Read this chapter for a description of the ARM C and C++ libraries and instructions on re-implementing individual library functions. The ARM C libraries provide additional components to enable support for C++ and to compile code for different architectures and processors.

> For a description of the Rogue Wave Standard C++ Library, see the Rogue Wave HTML documentation supplied with RVDK.

Chapter 6 *Floating-point Support*

Read this chapter for a description of floating-point support in the ARM compiler and libraries.

Chapter 7 *Semihosting*

Read this chapter for information about the semihosting mechanism, which enables code running on an ARM target to use the I/O facilities on a host computer that is running an ARM debugger.

Appendix A *Via File Syntax*

Read this appendix for a description of the syntax for via files. You can use via files to specify command-line arguments to many ARM tools.

Appendix B *Standard C Implementation Definition*

Read this appendix for information on the ARM C implementation that relates directly to the ISO C requirements.

Appendix C *Standard C++ Implementation Definition*

Read this appendix for information on the ARM C++ implementation.

Appendix D *C and C++ Compiler Implementation Limits*

Read this appendix for implementation limits of C and C++ in the ARM compiler.

Appendix E *Using Older Options*

Read this appendix for details of changes to compiler option names. This appendix describes changes in this release, and 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*\RVDK\Examples\.... 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:

monospace Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

<u>mono</u>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.

italic   Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold   Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM processor signal names.

Further reading

This section lists publications from both ARM Limited and third parties 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 reference information that is specific to development tools supplied with RVDK. Other publications included in the suite are:

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

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

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

Other publications

This book is not intended to be an introduction to the C, or C++ programming languages. It does not try to teach programming in C or C++, and it is not a reference manual for the C or C++ standards. Other books provide general information about programming.

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.

The following books describe the C++ language:

*   *ISO/IEC 14882:1998(E), C++ Standard*. Available from national standards bodies.

*   Stroustrup, B., *The C++ Programming Language* (3rd edition, 1997). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-88954-4.

The following books provide general C++ programming information:

*   Stroustrup, B., *The Design and Evolution of C++* (1994). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-54330-3.

    This book explains how C++ evolved from its first design to the language in use today.

*   Vandevoorde, D and Josuttis, N.M. *C++ Templates: The Complete Guide* (2003). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-73484-2.

*   Meyers, S., *Effective C++* (1992). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-56364-9.

    This provides short, specific, guidelines for effective C++ development.

*   Meyers, S., *More Effective C++* (2nd edition, 1997). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-92488-9.

The following books provide general C programming information:

*   ISO/IEC 9899:1990, *C Standard*.

    This is available from ANSI as X3J11/90-013. The standard is available from national standards bodies (for example, AFNOR in France, ANSI in the USA).

*   Kernighan, B.W. and Ritchie, D.M., *The C Programming Language* (2nd edition, 1988). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-110362-8.

This is the original C bible, updated to cover the essentials of ANSI C.

*   Harbison, S.P. and Steele, G.L., *A C Reference Manual* (5th edition, 2002). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-089592-X.

    This is a very thorough reference guide to C, including useful information on ANSI C.

*   Plauger, P., *The Standard C Library* (1991). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-131509-9.

    This is a comprehensive treatment of ANSI and ISO standards for the C Library.

*   Koenig, A., *C Traps and Pitfalls,* Addison-Wesley (1989), Reading, Mass. ISBN 0-201-17928-8.

    This explains how to avoid the most common traps in C programming. It provides informative reading at all levels of competence in C.

See `http://dwarf.freestandards.org` for information about the DWARF2 and DWARF3 debug table standards.

Feedback

ARM Limited welcomes feedback on both RVDK and the documentation.

Feedback on RealView Developer Kit

If you have any problems with RVDK, 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 tools, including the version number and build numbers.

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 the ARM® compiler provided with *RealView® Developer Kit* (RVDK). It contains the following sections:

- *About RVDK* on page 1-2
- *The ARM compiler and libraries* on page 1-3.

1.1 About RVDK

RVDK consists of a suite of applications, together with supporting documentation and examples, that enable you to write applications for the ARM family of *Reduced Instruction Set Computing* (RISC) processors. You can use RVDK to build C, C++, and ARM assembly language programs.

——— **Note** ———

Support for C++ may be obtained as an upgrade on certain versions of RVDK, by means of an appropriate FLEX*lm* license.

The RVDK toolkit consists of the following major components:
* command-line development tools
* utilities
* supporting software
* RealView Debugger.

This book describes the ARM compiler, armcc, and the libraries provided with RVDK. See *ARM publications* on page x for a list of the other books in the RVDK documentation suite that give information on the ARM assembler, compiler, and supporting software.

1.1.1 Using the examples

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

1.2   The ARM compiler and libraries

This section gives an overview of the ARM compiler, and the C and C++ libraries:

*   *The ARM compiler*
*   *The C and C++ libraries* on page 1-4
*   *Compatibility with legacy objects and libraries* on page 1-4.

———— **Note** ————

The compiler requires the RealView ICE Micro Edition (RVI-ME) that is supplied with your copy of RVDK to be plugged into the USB port of your computer in order to work. The compiler does not operate if the correct RVI-ME is not detected.

1.2.1   The ARM compiler

The ARM compiler, `armcc`:

*   Compiles:
    —   ISO Standard C:1990 source
    —   ISO Standard C++:1998 source

    into:

    —   32-bit ARM code
    —   16-bit Thumb code.

*   Is an optimizing compiler. Command-line options enable you to control the level of optimization. See *Defining optimization criteria* on page 2-43 for details.

*   Generates output objects in ELF format and DWARF2 debug information. *RealView®️ Compilation Tools* (RVCT) v2.2 also contains initial support for DWARF3 (Draft Standard 9) debug tables. In addition, the ARM compiler can generate an assembly language listing of the output code, and can interleave an assembly language listing with source code.

*   Complies with the *Application Binary Interface (ABI) for the ARM Architecture (base standard)* [BSABI]. See *ABI for the ARM Architecture compliance* on page 2-3 for more details.

RVDK uses an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The file format for each version of RVDK is restricted to the proprietary ATPE format for the permitted device. This is referred to as *ATPE_Custom*.

See Chapter 2 *Using the ARM Compiler* for more information on the ARM compiler.

1.2.2 The C and C++ libraries

RVDK provides the following runtime C and C++ libraries:

The ARM C libraries

> The ARM C libraries provide standard C functions, and helper functions used by the C and C++ libraries. The C libraries also provide target-dependent functions that are used to implement the standard C library functions in a semihosted environment. The C libraries are structured so that you can redefine target-dependent functions in your own code to remove semihosting dependencies.

Rogue Wave Standard C++ Library version 2.02.03

> The Rogue Wave Standard C++ Library, as supplied by Rogue Wave Software, Inc., provides standard C++ functions and objects such as `cout`. It includes data structures and algorithms known as the *Standard Template Library* (STL). The C++ libraries use the C libraries to provide target-specific support. The Rogue Wave Standard C++ Library is provided with C++ exceptions enabled.

> For more information on the Rogue Wave libraries, see the Rogue Wave HTML documentation and the Rogue Wave web site at `http://www.roguewave.com/`.

Support libraries

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

The C and C++ libraries are provided as binaries only. There is a variant of the 1990 ISO Standard C library for each combination of major build options, such as the byte order of the target system, whether interworking is selected, and whether floating point support is selected. See Chapter 5 *The C and C++ Libraries* for more information on the libraries.

1.2.3 Compatibility with legacy objects and libraries

By default, objects generated by the RVCT v2.x C++ compiler are not binary-compatible with those generated by the *ARM Developer Suite*™ (ADS) v1.2 and RVCT v1.2 C++ compilers.

——— **Note** ———

To avoid potential incompatibilities, and to benefit from the improved optimization and new features, it is recommended that you rebuild your entire project, including your own libraries, with the latest release of RVDK.

However, if you require binary compatibility with objects built with ADS v1.2 or RVCT v1.2, see *ADS ABI qualifier* on page 2-64 for more information.

ARM DUI 0282B

Chapter 2
Using the ARM Compiler

This chapter describes the command-line options to the ARM® compiler, `armcc`. It contains the following sections:

- *About the ARM compiler* on page 2-2
- *File usage* on page 2-10
- *armcc command syntax* on page 2-20.

———— **Note** ————

The compiler requires the RealView ICE Micro Edition (RVI-ME) that is supplied with your copy of RVDK to be plugged into the USB port of your computer in order to work. The compiler does not operate if the correct RVI-ME is not detected.

2.1 About the ARM compiler

The ARM compiler, `armcc`, enables you to compile your ARM and Thumb® C and C++ code. The compiler delivered with *RealView® Compilation Tools* (RVCT) since v2.0 provides the functionality of the earlier ARM and Thumb C and C++ compilers in a single executable.

This section describes:

- *Standards and compatibility*
- *Invoking the ARM compiler* on page 2-3
- *Multifile compilation* on page 2-6
- *Linker feedback* on page 2-7
- *Inline and embedded assembly language* on page 2-8
- *Semihosting* on page 2-8
- *Library support* on page 2-9.

RVDK uses an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The file format for each version of RVDK is restricted to the proprietary ATPE format for the permitted device. This is referred to as *ATPE_Custom*.

2.1.1 Standards and compatibility

This section describes the standards to which the RVCT compiler conforms, and compatibility with legacy objects.

Source language modes

The ARM compiler has two distinct source language modes that you can use to compile different varieties of C and C++ source code:

ISO C   The ARM compiler compiles C as defined by ISO/IEC 9899:1990 (E), including its Technical Corrigendums 1 & 2. Some features of C99 are also available, that is `long long` (see *long long* on page 3-33) and `restrict` (see *restrict* on page 3-34).

The ARM compiler supports the additions to C90 in Normative Addendum 1, that is `wchar.h` and `wctype.h`, added in 1994.

The compiler is tested against version 04a of *The Plum Hall C Validation Suite* . The compiler option `--strict` is used when running the tests.

ISO C++   The ARM compiler expects C++ that conforms to the ISO/IEC 14822 :1998 International Standard for C++. See Appendix C *Standard C++ Implementation Definition* for a detailed description of ARM C++ language support.

The compiler is tested against version 03a of *Suite++: The Plum Hall Validation Suite for C++*. The compiler option --strict is used when running the tests.

For more information on how to use compiler options to set the source mode for the compiler, see *Setting the source language* on page 2-27.

ABI for the ARM Architecture compliance

The *ABI for the ARM Architecture (base standard)* [BSABI] enables you to use ARM and Thumb objects and libraries from different producers that support ABI for the ARM Architecture. RVCT fully supports the BSABI, including initial support for DWARF3 (Draft Standard 9) debug tables.

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

——— **Note** ———

Support for C++ may be obtained as an upgrade on certain versions of RVDK, by means of an appropriate FLEX*lm* license.

2.1.2   Invoking the ARM compiler

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. You can use the POSIX option -- to specify that all subsequent arguments are not treated as command switches. For example, to compile a file named -ifile_1 type:

```
armcc -O2 -- -ifile_1
```

Invoking the ARM compiler using older tool names

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

Table 2-1 Start-up configuration based on old tool names

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

Default behavior

Table 2-2 shows how the compiler start-up configuration is adjusted by the filename extension you specify.

Table 2-2 Start-up configuration as adjusted by filename extension

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

When you compile multiple files with a single command, all files must of the same type (either C or C++). The compiler cannot switch the language based on the file extension. The following example produces an error, because the compiler attempts to compile using the source language of test1.c:

```
armcc -c test1.c test2.cpp
Error: C3472E: No source language specified (--cpp or --c90) and sourcefile
extensions conflict
```

If you specify files with conflicting file extensions you can force the compiler to compile either for C or for C++, regardless of file extension. For example:

```
armcc -c --cpp test1.c test2.cpp
```

See *Setting the source language* on page 2-27 for details.

Support for processing *PreCompiled Header* (PCH) files is not available when you specify multiple source files in a single compilation. If you request PCH processing and specify more than one primary source file, the compiler issues an error message, and aborts the compilation. See *Precompiled header files* on page 2-14 for details.

Overriding the default behavior

You can override the adjustments that the ARM compiler makes based on the filename extension (see Table 2-2 on page 2-4) or the tool name you used to invoke the compiler (see Table 2-1 on page 2-4). To do this, use the appropriate command-line option as shown in Table 2-3. See *Pragmas controlling code generation* on page 3-5 for information on using pragmas to set source language.

Table 2-3 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, if you submit the following command:

```
tcpp foo.acpp --c90
```

The compiler makes determinations as shown in Table 2-4.

Table 2-4 Example configuration

Command Component	Description	Defines instruction set	Defines source language
tcpp	Tool name	Thumb	C++
.acpp	Filename extension	ARM	C++
--c90	Command-line option	No adjustment	C
	Result	ARM	C

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

To summarize:

• the filename extension overrides the default configuration determined by the tool name used to invoke the ARM compiler

• the command-line option overrides the default configuration determined by the filename extension.

See *Setting the source language* on page 2-27 and *Targeting the instruction set* on page 2-43 for details on these options.

2.1.3 Multifile compilation

Multifile compilation enables the compiler to perform optimization across multiple files instead of on individual files. You can specify multifile compilation using the --multifile optimization option, for example:

```
armcc -c --multifile test1.c ... testn.c
```

The specified files are compiled into a combined object file and dummy object files. The combined object file is named after the first source file you specify, in this example that is test1.o.

An empty object file is created for each subsequent source file you specify, (in this example, test2.o to testn.o) to meet the requirements of standard make systems.

If you want to specify a different name for the combined object file, use the -o *filename* option, for example:

```
armcc -c --multifile test1.c ... testn.c -o test.o
```

You can also use the --default_extension option to specify a default extension for your object files. See *Default object extension* on page 2-57 for more details.

For more details on the --multifile optimization option, see *Multi-optimization options* on page 2-43.

2.1.4   Linker feedback

Linker feedback enables the efficient elimination of unused functions. Unused function code might occur in the following situations:

- You might have legacy functions that are no longer used in your sources. Rather than remove the unused function code from your sources, you can use linker feedback to remove the unused code from the final image.

- When a function is inlined by the compiler or by the linker (using the --inline optimization option). If an inlined function is not declared as **static**, the out-of-line function code is still present in the object file, but there is no longer a call to that code.

- Where your image contains both ARM and Thumb code, functions that are called from the other state must be built for interworking. You can use linker feedback to avoid compiling for interworking those functions that are never called from the other state.

To provide linker feedback to the compiler, you must link your code as a separate step, and use the armlink option --feedback *filename* to create a feedback file. You can then use this file as an input to the compiler using the compiler option --feedback *filename*. The feedback file lists any unused functions.

You can specify the --feedback *filename* option even when no feedback file exists. This enables you to create a build file without having to modify it again, for example:

```
armcc -c --feedback unused.txt test.c -o test.o
armlink --feedback unused.txt test.o -o test.axf
```

The first time you build the application, the compiler warns you that it cannot read the specified feedback file (because it does not exist). The link command then creates the feedback file and builds the image. Each subsequent compile step uses the feedback file from the previous link step to remove any unused functions that are identified.

You can perform multiple compilations using the same feedback file, or specify multiple source files in a single compilation. The feedback file identifies the source file that contains the unused functions in a comment, and is not used by the compiler. The compiler places each unused function identified in the feedback file into its own ELF section in the corresponding object file.

Note

To get the maximum benefit from linker feedback you have to do a full compile and link at least twice. However, a single compile and link using feedback from a previous build is usually sufficient.

Because the feedback file contains information about a previous build, it might be out of date. That is, a function previously identified as being unused might be used in the current source. The linker removes the code for an unused function only if it is not used in the current source. For this reason, linker feedback is a safe optimization, but there might be a small impact on code size.

See *Multi-optimization options* on page 2-43 for more details on the --feedback option. Also, see *RealView Developer Kit v2.2 Linker and Utilities Guide* for details on using armlink with feedback.

2.1.5   Inline and embedded assembly language

The ARM compiler enables you to include assembly code in your C and C++ sources. This can range from small fragments of assembly code inside C or C++ functions, known as *inline assembly*, to complete functions that are outside of other C or C++ functions, known as *embedded assembly*.

See Chapter 4 *Inline and Embedded Assemblers* for full details on the inline and embedded assemblers.

2.1.6   Semihosting

Development hardware often does not have all the input and output facilities of the final system. Therefore, a software mechanism is required to service input/output (I/O) requests from application code. This mechanism is implemented using *software interrupt* (SWI) operations, and is called *semihosting*.

Semihosting enables ARM targets to communicate I/O requests from application code to a host computer that is running a debugger. Therefore, you can use the I/O facilities of the debugger on your host computer instead of providing the facilities on your target system.

C and C++ code uses semihosting facilities by default. The semihosting SWI is usually invoked by code within library functions. However, the application can also invoke the semihosting SWI directly.

To access semihosting facilities from assembly code, use semihosting SWIs. Any of the following intercept semihosting SWIs and request services from the host computer:

*   RealView® ICE Micro Edition (RVI-ME)
*   a debug monitor.

For more details about semihosting, see Chapter 7 *Semihosting*.

2.1.7   Library support

RVCT provides both ISO C libraries and Rogue Wave C++ libraries. Both sets of libraries are in prebuilt binary form. See Chapter 5 *The C and C++ Libraries* for detailed information about the libraries.

You can create your own definition of target-dependent functions to customize the C libraries. Processor-specific retargeting is done automatically by setting the compiler options for the processor architecture and family.

2.2 File usage

This section describes how files are used in the ARM compiler, armcc:

• *Naming conventions*
• *Included files* on page 2-11
• *Precompiled header files* on page 2-14.

2.2.1 Naming conventions

The ARM compiler uses suffix naming (filename-extension) conventions to identify the classes of file involved in compilation and in the link stage. The names used on the command line, and as arguments to preprocessor #include directives, map directly to host filenames under Windows/MS-DOS.

The ARM compiler uses or generates files with the following file suffixes:

filename.c	The ARM compiler recognizes .c, .cpp, .cp, .c++, .ac, .acpp, .tc, and .tcpp suffixes as source files. Also accepts suffixes containing numbers, that is .c*nnn*, .ac*nnn*, and .tc*nnn* (recognized as .c).
filename.cc	Implicit include file (see *Implicit inclusion* on page 3-60.
filename.d	Dependency list file (the default output extension for the --md option).
filename.h	Header file (a convention only, this suffix has no special significance for the compiler).
filename.o	ARM object file in ELF format. Also accepts .o*nnn* (recognized as .o).
filename.s	ARM or Thumb assembly language file. You can use this in the input file list. If you do, the compiler invokes the assembler, armasm, to assemble the file.
filename.lst	Error and warning list file (the default output extension for the --list option).

Portability

The ARM compiler supports multiple file naming conventions on all supported hosts. To ensure portability between hosts, use the following guidelines:

• Ensure that filenames do not contain spaces. If you have to use path names or filenames containing spaces, enclose the path and filename in quotes.

- Make embedded path names relative rather than absolute.

In each host environment, the ARM compiler supports:

- native filenames

- pseudo UNIX filenames in the format:

  `host-volume-name:/rest-of-unix-file-name`

- UNIX filenames using / as a path separator.

Filenames are parsed as follows:

- a name starting with `host-volume-name:/` is a pseudo UNIX filename

- a name that does not start with `host-volume-name:/` and contains / is a UNIX filename

- a name that does not contain a / is a host filename.

Filename validity

The ARM compiler does not check that filenames are acceptable to the host file system. If a filename is not acceptable, the compiler reports that the file cannot be opened.

Output files

By default, the output files created by an ARM compiler are stored in the current directory. Object files are written in *ARM Executable and Linkable Format* (ELF).

RVDK uses an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The file format for each version of RVDK is restricted to the proprietary ATPE format for the permitted device. This is referred to as *ATPE_Custom*.

2.2.2 Included files

Several factors affect the way the ARM compiler searches for #include header files and source files. These include:

- the -I and -J compiler options

- the --kandr_include and --sys_include compiler options

- the value of the environment variable RVCT22INC or, if this variable is not set, ARMINC

- whether the filename is an absolute filename or a relative filename

---

•   whether the filename is between angle brackets or double quotes.

The current place

By default, the ARM compiler uses Berkeley UNIX search rules, so source files and
#include header files are searched for relative to the *current place*. This is the directory
containing the source or header file currently being processed by the compiler.

When a file is found relative to an element of the search path, the directory containing
that file becomes the new current place. When the compiler has finished processing that
file, it restores the previous current place. At each instant there is a stack of current
places corresponding to the stack of nested #include directives. For example, if the
current place is the RVCT include directory ...\include, and the compiler is seeking
the include file sys\defs.h, it locates ...\include\sys\defs.h if it exists.

When the compiler begins to process defs.h, the current place becomes
...\include\sys. Any file included by defs.h that is not specified with an absolute path
name, is searched for relative to ...\include\sys.

The original current place ...\include is restored only when the compiler has finished
processing defs.h.

You can disable the stacking of current places by using the compiler option
--kandr_include. This option makes the compiler use the search rule originally
described by Kernighan and Ritchie in *The C Programming Language*. Under this rule
each nonrooted user #include is searched for relative to the directory containing the
source file that is being compiled.

The RVCT22INC environment variable

The RVCT22INC environment variable points to the location of the included header and
source files provided with RVCT. You must not change this environment variable. If you
want to include files from other locations, use the -I and -J command-line options as
required. These options control the search order for header files. See *Include file options*
on page 2-29 for more details.

When compiling, directories specified with RVCT22INC are searched immediately after
directories specified by the -I option have been searched. If you use the -J option,
RVCT22INC is ignored.

——— **Note** ———
If RVCT22INC is not set, then the ARMINC environment variable is used.

The search path

Table 2-5 shows how the command-line options affect the search path used by the compiler when it searches for included header and source files.

Table 2-5 Include file search paths

Compiler option	<include> search order	"include" search order
Neither -I nor -J	RVCT22INCdirs	CP, RVCT22INCdirs
-J	J*dirs*	CP, and J*dirs*
-I	RVCT22INCdirs, I*dirs*	CP, I*dirs*, RVCT22INCdirs
Both -I and -J	J*dirs*, I*dirs*	CP, I*dirs*, J*dirs*
--sys_include	No effect	Removes CP from the search path
--kandr_include	No effect	Uses Kernighan and Ritchie search rules

In this table:

RVCT22INC**dirs**

> The list of directories specified by the RVCT22INC environment variable, if it is set. If this is not set, the compiler searches the list of directories specified by the ARMINC environment variable.

CP   The current place (see *The current place* on page 2-12 for more details).

I*dirs* and J*dirs*

> The directories specified by the -I*dirs* and -J*dirs* compiler options. If your application and system headers are in the same directory, for example, C:\myincludes, you can include the path in the -J option, and do not have to use -I. For example:
>
> -J"c:\myincludes,$RVCT22INC"
>
> You must enclose this in double quotes, because the default path defined by RVCT22INC contains spaces.

The TMP and TMPDIR environment variables

On Windows platforms, the environment variable TMP is used to specify the directory to be used for temporary files. If TMP is not set, then TMPDIR is used.

2.2.3   Precompiled header files

When you compile your source files, the included header files are also compiled. If a header file is included in more than one source file, it is recompiled when each source file is compiled. Also, you might include header files that introduce many lines of code, but the primary source files that include them are relatively small. Therefore, it is often desirable to avoid recompiling a set of header files by precompiling them. These are referred to as *PreCompiled Header* (PCH) files.

By default, when the compiler creates a PCH file, it:

* takes the name of the primary source file and replaces the suffix with `.pch`
* creates the file in the same directory as the primary source file.

─────── **Note** ───────

Support for PCH processing is not available when you specify multiple source files in a single compilation. If you request PCH processing and specify more than one primary source file, the compiler issues an error message, and aborts the compilation.

───────────────────────

The ARM compiler can precompile header files automatically, or enable you to control the precompilation. For more information, see:

* *Automatic PCH processing*
* *Manual PCH processing* on page 2-17
* *Controlling the output of messages during PCH processing* on page 2-18
* *Performance issues* on page 2-18.

Automatic PCH processing

When you use the `--pch` command-line option, automatic PCH processing is enabled. This means that the compiler automatically looks for a qualifying PCH file, and reads it if found. Otherwise, the compiler creates one for use on a subsequent compilation.

When the compiler creates a PCH file, it takes the name of the primary source file and replaces the suffix with `.pch`. The PCH file is created in the directory of the primary source file, unless you specify the `--pch_dir` option (see *PCH processing options* on page 2-30 for details).

The header stop point

The PCH file contains a snapshot of all the code that precedes a *header stop point*. Typically, the header stop point is the first token in the primary source file that does not belong to a preprocessing directive. In the following example, the header stop point is `int` and the PCH file contains a snapshot that reflects the inclusion of `xxx.h` and `yyy.h`:

```
#include "xxx.h"
#include "yyy.h"
int i;
```

——— **Note** ———

You can manually specify the header stop point with `#pragma hdrstop`. You must place this before the first token that does not belong to a preprocessing directive. In this example, place it before `int`. See *Controlling PCH processing* on page 2-17 for more details.

_____

Conditions that affect PCH file generation

If the first non-preprocessor token, or a `#pragma hdrstop`, appears within a `#if` block, the header stop point is the outermost enclosing `#if`. For example:

```
#include "xxx.h"
#ifndef YYY_H
#define YYY_H 1
#include "yyy.h"
#endif
#if TEST
int i;
#endif
```

In this example, the first token that does not belong to a preprocessing directive is `int`, but the header stop point is the start of the `#if` block containing it. The PCH file reflects the inclusion of `xxx.h` and, conditionally, the definition of `YYY_H` and inclusion of `yyy.h`. It does not contain the state produced by `#if TEST`.

A PCH file is produced only if the header stop point and the code preceding it (mainly, the header files) meet the following requirements:

*   The header stop point must appear at file scope. It must not be within an unclosed scope established by a header file. For example, a PCH file is not created in this case:

    ```
    // xxx.h
    class A {
    // xxx.c
    #include "xxx.h"
    int i; };
    ```

- The header stop point must not be inside a declaration that is started within a header file. Also, in C++, it must not be part of a declaration list of a linkage specification. For example, in the following case the header stop point is int, but because it is not the start of a new declaration, no PCH file is created:

```
// yyy.h
static
// yyy.c
#include "yyy.h"
int i;
```

- The header stop point must not be inside a #if block or a #define that is started within a header file.

- The processing that precedes the header stop point must not have produced any errors.

  ———— **Note** ————

  Warnings and other diagnostics are not reproduced when the PCH file is reused.

- No references to predefined macros __DATE__ or __TIME__ must appear.

- No instances of the #line preprocessing directive must appear.

- #pragma no_pch must not appear (see *Pragmas controlling PCH processing* on page 3-8).

- The code preceding the header stop point must have introduced a sufficient number of declarations to justify the overhead associated with precompiled headers.

More than one PCH file might apply to a given compilation. If so, the largest is used, that is, the one representing the most preprocessing directives from the primary source file. For instance, a primary source file might begin with:

```
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
```

If there is one PCH file for xxx.h and a second for xxx.h and yyy.h, the latter PCH file is selected, assuming that both apply to the current compilation. Also, after the PCH file for the first two headers is read in and the third is compiled, a new PCH file for all three headers might be created.

In automatic PCH processing mode the compiler indicates that a PCH file is obsolete, and deletes it, under the following circumstances:

- if the PCH file is based on at least one out-of-date header file but is otherwise applicable for the current compilation

- if the PCH file has the same base name as the source file being compiled (for example, xxx.pch and xxx.c) but is not applicable for the current compilation (for example, because you have used different command-line options).

These describe some common cases. Other PCH files must be deleted by you.

Manual PCH processing

You can specify the filename and location of PCH files, and the parts of a header file that are subject to PCH processing. See the following sections for more details:
- *Specifying a PCH filename and location*
- *Controlling PCH processing*.

Specifying a PCH filename and location

You can specify the filename and location of the PCH file using the following command-line options:
- `--create_pch` *filename*
- `--use_pch` *filename*
- `--pch_dir` *directory*

If you use either `--create_pch` or `--use_pch` with the `--pch_dir` option, the indicated filename is appended to the directory name, unless the filename is an absolute path name.

You must not use the options `--create_pch`, `--use_pch`, and `--pch` together. If more than one of these options is specified, only the last option applies. Nevertheless, most of the description of automatic PCH processing applies to one or the other of these modes. For example, header stop points and PCH file applicability are determined in the same way.

See *PCH processing options* on page 2-30 for more details on these options.

Controlling PCH processing

You can specify the parts of a header file are subject to PCH processing using the following pragmas:

- Insert a manual header stop point using the `#pragma hdrstop` directive in the primary source file before the first token that does not belong to a preprocessing directive.

This enables you to specify where the set of header files that are subject to precompilation ends. For example,

```
#include "xxx.h"
#include "yyy.h"
#pragma hdrstop
#include "zzz.h"
```

In this example, the PCH file includes the processing state for xxx.h and yyy.h but not for zzz.h. This is useful if you decide that the information following the #pragma hdrstop does not justify the creation of another PCH file.

• Use the #pragma no_pch directive to suppress PCH processing for a source file.

See *Pragmas controlling PCH processing* on page 3-8 for more details on these pragmas.

——— **Note** ———

You can use these pragmas even if you are using automatic PCH processing, see *Automatic PCH processing* on page 2-14.

Controlling the output of messages during PCH processing

When the compiler creates or uses a PCH file, it displays the following message:

```
test.c: creating precompiled header file test.pch
```

You can suppress this message by using the command-line option --no_pch_messages.

When you use the --pch_verbose option, the compiler displays a message for each PCH file that is considered, but cannot be used. The compiler also gives the reason why it cannot be used.

Performance issues

Typically, the overhead of creating and reading a PCH file is small even for reasonably large header files, and even if the created PCH file is not used. If the file is used, there is typically a significant decrease in compilation time. However, PCH files can range in size from about 250KB to several megabytes or more, so you might not want many PCH files created.

PCH processing might not always be appropriate, for example, where you have an arbitrary set of files with non-uniform initial sequences of preprocessing directives.

The benefits of PCH processing occur when several source files can share the same PCH file. The more sharing, the less disk space is consumed. Sharing minimizes the disadvantage of large PCH files, without giving up the advantage of a significant decrease in compilation times.

Therefore, to take full advantage of header file precompilation, you might have to re-order the `#include` sections of your source files, or group `#include` directives within a commonly used header file.

Different environments and different projects might have differing requirements. Be aware, however, that making the best use of PCH support might require some experimentation and probably some minor changes to source code.

2.3 armcc command syntax

This section describes the command syntax for the ARM compiler:

- *Command-line options*
- *Invoking the ARM compiler* on page 2-23
- *Setting the source language* on page 2-27
- *Specifying search paths* on page 2-28
- *PCH processing options* on page 2-30
- *Setting preprocessor options* on page 2-31
- *Syntax-checking* on page 2-32
- *C++ Language configuration and object generation* on page 2-32
- *Specifying output format* on page 2-37
- *Specifying the target processor or architecture* on page 2-39
- *Generating debug information* on page 2-41
- *Controlling code generation* on page 2-42
- *Adding symbol versions* on page 2-56
- *Default object extension* on page 2-57
- *Diagnostic messages* on page 2-58
- *Using AAPCS options* on page 2-64.

RVDK uses an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The file format for each version of RVDK is restricted to the proprietary ATPE format for the permitted device. This is referred to as *ATPE_Custom*.

Compilation tools only support the permitted ARM CPU for a given RVDK toolkit.

2.3.1 Command-line options

You can control many aspects of compiler operation with command-line options. The ARM compiler uses the *Edison Design Group* (EDG) front end. See the Edison Design Group web site at `http://www.edg.com` for more information on the EDG front end.

The following rules apply, depending on the type of option:

Single-letter options

All single-letter options, or single-letter options with arguments, are preceded by a single dash -. You can use a space between the option and the argument, or the argument can immediately follow the option. For example:

```
-J directory
-Jdirectory
```

Keyword options

> All keyword options, or keyword options with arguments, are preceded
> by a double dash --. A space or = character is required between the option
> and the argument. For example:

```
--depend file.d
--depend=file.d
```

Compiler options that contain non-leading - or _ can use either of these characters. For
example, --force_new_nothrow is the same as --force-new-nothrow.

You can also use the POSIX option --. After this all strings count as non-option
parameters. For example:

```
armcc -02 -- -ifile_1
```

Ordering command-line options

In general, command-line options can appear in any order in a single compiler
invocation. However, the effects of some options depend on the order they appear in the
command line, for example optimization options prefixed by -0.

The compiler enables you to use multiple options even where these might conflict. This
means that you can append new options to an existing command line, for example, in a
make file or via file.

Where options override previous options on the same command line, the last one found
always takes precedence. For example:

```
armcc -01 -02 -0space -0time ...
```

is executed by the compiler as:

```
armcc -02 -0time
```

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.

Specifying command-line options with an environment variable

You can specify command-line options by setting the value of the RVCT22_CCOPT environment variable. The syntax is identical to the command line syntax. The compiler reads the value of RVCT22_CCOPT and inserts it at the front of the command string. This means that options specified in RVCT22_CCOPT can be over-ridden by arguments on the command-line.

Autocompletion of command-line options

You can optionally request the autocompletion of command-line options. To do this, place a dot (.) after the characters to be autocompleted. Autocompletion only applies to keyword options (see *Command-line options* on page 2-20).

Arguments must be separated from the dot by a space or = character. You cannot use autocompletion for the arguments to an option.

You must include sufficient characters to make the autocompleted option unique. For example, to specify --exceptions_unwind, the minimum characters you have to use is:

```
--exceptions_.
```

2.3.2 Invoking the ARM compiler

The command for invoking the ARM compiler is:

```
armcc [source-language] [search-paths] [PCH-options] [preprocessor-options]
[syntax-checking]  [C++-language] [output-format] [target-options]
[debug-options] [code-generation-options]  [default-object-extension]
[diagnostic-options] [warning-options] [additional-checks] [error-options]
[PCS-options] [source]
```

In general, the command-line options can appear in any order. However, the effects of some options depend on the order they appear in the command line.

The ARM compiler options are:

source-language	This specifies the variant of source language that is accepted by the compiler. The default is ISO C when compiling C code and ISO C++ when compiling C++ code. See *Setting the source language* on page 2-27 for details.
search-paths	This specifies the directories that are searched for included files. See *Specifying search paths* on page 2-28 for details.
PCH-options	This specifies the processing of PCH files. See *PCH processing options* on page 2-30 for details.
preprocessor-options	
	This specifies preprocessor behavior, including preprocessor output and macro definitions. See *Setting preprocessor options* on page 2-31 for details.
syntax-checking	This specifies that syntax-checking is to be performed on the source code. See *Syntax-checking* on page 2-32 for details.
C++-language	This specifies options specific to C++ compilation. See *C++ Language configuration and object generation* on page 2-32 for details.
output-format	This specifies the format for the compiler output. You can use these options to generate assembly language output listing files and object files. See *Specifying output format* on page 2-37 for details.
target-options	This specifies the target processor or architecture. See *Specifying the target processor or architecture* on page 2-39 for details.

debug-options	This specifies whether or not debug tables are generated, and their format. See *Generating debug information* on page 2-41 for details.
code-generation-options	
	This specifies options such as optimization, byte order, and alignment of data produced by the compiler. See *Controlling code generation* on page 2-42 for details.
default-object-extension	
	This specifies the default object extension to use. See *Default object extension* on page 2-57 for details.
diagnostic-options	This specifies options that enable you to control the diagnostic messages that are output by the compiler. See *Diagnostic messages* on page 2-58 for details.
warning-options	This specifies whether specific warning messages are generated. See *Suppressing warning messages with the -W option* on page 2-62 for details.
additional-checks	This specifies additional checks that can be applied to your code, such as checks for data flow anomalies and unused declarations. See *Data flow warnings* on page 2-63 for details.
error-options	This enables you to turn off specific recoverable errors or downgrade specific errors to warnings. See *Changing the severity of diagnostic messages* on page 2-61 for details.
PCS-options	This specifies the procedure call standard to use. See *Using AAPCS options* on page 2-64 for details.
source	This provides the filenames of one or more text files containing C or C++ source code. By default, the compiler looks for source files, and creates output files, in the current directory.
	If a source file is an assembly file (that is, one with an extension of .s) the compiler activates the ARM assembler to process the source file.

——— **Note** ———

If you are compiling multiple source files, see *Multifile compilation* on page 2-6 for more details.

For details on the compiler language that is assumed when you invoke armcc, see *Invoking the ARM compiler* on page 2-3.

Reading compiler options from a file

When the operating system restricts the command line length, you can include additional command-line options in a file with the compiler option:

`--via` *filename*

The compiler opens the specified file and reads the additional command-line options from it.

You can nest `--via` calls within via files, for example, by including `--via` *filename2* in the file. In the following example, the options specified in `input.txt` are read as the command line is parsed:

```
armcc --via input.txt source.c
```

See Appendix A *Via File Syntax* for more information on writing via files.

Specifying keyboard input

Use minus – as the source filename to instruct the compiler to take input from the keyboard. The default compiler mode is C with the `armcc` command, and C++ with the `armcpp` command. To terminate input, enter `Ctrl-Z` then `Return` on Microsoft Windows systems.

An assembly listing for the keyboard input is sent to the output stream after input has been terminated if both the following are true:
- no output file is specified
- no preprocessor-only option is specified, for example `-E`.

If you specify an output file with the `-o` option, an object file is written. If you specify the `-E` option, the preprocessor output is sent to the output stream. If you specify the `-o-` option, the output is sent to the console.

Getting help and version information

Use the `--help` option to view a summary of the main compiler command-line options.

Use the `--vsn` option to display the version string for the compiler. This is the default if you do not specify any options or source files.

Redirecting diagnostics

Use the `--errors` *filename* option to redirect compiler diagnostic output to a file. Diagnostics that relate to the command options are not redirected. See *Controlling the output of diagnostic messages* on page 2-60 for more details.

 ARM DUI 0282B

2.3.3 Setting the source language

This section describes options that determine the source language variant accepted by the compiler (see also *Controlling code generation* on page 2-42).

These options enable you to specify the compilation language used by the compiler, and to determine how strictly the compiler enforces the standards and conventions of that language. By default, the compiler compiles ISO C code. For C++ code, the compiler compiles as much as it can of ISO/IEC C++. For details on the default language assumed by the compiler, based on filename extension, see *Invoking the ARM compiler* on page 2-3.

The following options are used for setting the source language:

--cpp   Enables compilation of ISO/IEC C++. This is the default for `*.cpp` files.

--c90   Enables compilation of C rather than C++, specifically the C90 ISO version of C. This is the default for `*.c` files.

                The default mode is a fairly strict ISO compiler, but without some of the inconvenient features of the ISO C Standard. Some minor extensions are also supported, for example // in comments and $ in identifiers.

--gnu   Enables or disables the GNU compiler extensions that are supported by the ARM compiler. For details of the supported GNU extensions, see *GNU extensions to the ARM compiler* on page 3-66.

--no_strict   Relaxes the ISO language conformance. This is the default.

--strict --strict_warnings

                Enables strict ISO mode (ISO/IEC 9899:1990 (E)) and the C++ Standard (ISO/IEC 14882:1998 (E)). Diagnostic messages are returned when nonstandard features are used, and features that conflict with ISO C or C++ are disabled. This is compatible in both C and C++ mode.

                ISO violations can be issued as either warnings or errors, depending on the command-line option:

                • The --strict option causes errors to be issued.

                • The --strict_warnings option produces warnings.

                The error threshold is set so that the requested diagnostics are returned (see *Diagnostic messages* on page 2-58).

                For example, the following code segment returns an error when compiled with --cpp --strict, but only a warning with --cpp:

                ```
                static struct T {int i; };
                ```

Because no object is declared, static is spurious. Therefore, in the C++ Standard, this code segment is invalid.

--anachronisms --no_anachronisms

Enables or disables anachronisms in C++ mode. This option is valid only in C++ mode. See *Anachronisms* on page 3-59 for a complete description of anachronisms.

The default for this option is --no_anachronisms.

You can combine language options:

armcc --c90   Compiles ISO standard C, and the C mode of C++. This is the default.

armcc --strict   Compiles strict ISO standard C.

armcc --c90 --strict

Compiles strict ISO standard C (C mode of C++).

armcc --cpp   Compiles standard C++.

armcc --cpp --strict

Compiles strict C++.

armcc --c90 --gnu   Compiles ISO standard C, and the C mode of C++ with GNU extensions enabled (see *GNU extensions to the ARM compiler* on page 3-66).

armcc --cpp --gnu   Compiles standard C++ with GNU extensions enabled. By default, this disables exceptions handling (see *GNU extensions to the ARM compiler* on page 3-66).

2.3.4   Specifying search paths

These options enable you to specify the directories to search for included files.

Also, see *PCH processing options* on page 2-30 for a description of the options you can use to control the processing of PCH files.

                    ARM DUI 0282B

Include file options

The precise search path varies according to the combination of options you select, and whether the include file is enclosed in angle brackets or double quotes. See *Included files* on page 2-11 for full details of how these options work together:

`--preinclude` *filename*

Includes the source code of the specified file at the beginning of the compilation. This can be used to establish standard macro definitions, for example. The *filename* is searched for in the directories on the include search list.

`-I,` *directory*Adds the specified directory *directory*, or comma-separated list of directories, to the list of places that are searched to find included files. If you specify more than one directory, the directories are searched in the same order as the `-I` options specifying them.

See *Included files* on page 2-11 for complete details on how the compiler handles include files.

`--kandr_include`

Ensures that Kernighan and Ritchie search rules are used for locating included files. The current place is defined by the original source file and is not stacked. See *The current place* on page 2-12 for more information. If you do not use this option, Berkeley-style searching is used.

`--sys_include`

Removes the current place from the include search path. Quoted include files are treated in a similar way to angle-bracketed include files, except that quoted include files are always searched for first in the directories specified by `-I`, and angle-bracketed include files are searched for first in the `-J` directories (see Table 2-5 on page 2-13).

`-J`*directory*   Adds the specified *directory*, or comma-separated list of directories, to the list of system includes. RVCT22INC is set as the default system include unless you use `-J` to override it. Angle-bracketed include files are searched for first in the list of system includes, then any include list you have specified with `-I`.

See *Included files* on page 2-11 for complete details on how the compiler handles include files.

2.3.5   PCH processing options

These options enable you to control the processing of PCH files (see *Precompiled header files* on page 2-14 for more details):

--pch   Automatically uses or creates a PCH file.

If you include --use_pch or --create_pch (manual PCH mode) on the command line following this option, its effect is negated.

--create_pch `filename`

If other conditions are satisfied, this option creates a PCH file with the specified name.

If you include --pch (automatic PCH mode) or --use_pch on the command line following this option, its effect is negated.

--use_pch `filename`

Uses a PCH file of the specified `filename` as part of the current compilation.

If you include --pch (automatic PCH mode) or --create_pch on the command line following this option, its effect is negated.

--pch_dir `directory`

This option enables you to specify a `directory` to search for, or create, a PCH file.

You can use this option with automatic PCH mode (--pch) or with manual PCH mode (--create_pch or --use_pch).

--pch_messages --no_pch_messages

Enables or disables the display of a message indicating that a PCH file was created or used in the current compilation.

--pch_verbose

In automatic PCH mode, this option ensures that for each PCH file that cannot be used for the current compilation, a message is displayed giving the reason that the file cannot be used.

2.3.6   Setting preprocessor options

These options are used for controlling aspects of the preprocessor. (See *Pragmas* on page 3-2 for descriptions of other preprocessor options that can be set by pragmas.)

-E   Executes only the preprocessor phase of the compiler. By default, output from the preprocessor is sent to the standard output stream and can be redirected to a file using standard MS-DOS notation, for example:

```
armcc -E source.c > raw.c
```

You can also use the -o option to specify a file for the preprocessed output (see *Specifying output format* on page 2-37). By default, comments are stripped from the output. The preprocessor accepts source files with any extension (for example, .o, .s, and .txt). See also the -C option.

-C   Retains comments in preprocessor output when used in conjunction with -E, and must be specified after -E. The -C option does not request preprocessing output when used alone.

This option differs from the -c (lowercase) option that suppresses the link step. See *Specifying output format* on page 2-37 for a description of the -c option.

-M   Executes only the preprocessor phase of the compiler (see the -E option). This option produces a list of make file dependency lines suitable for use by a make utility. By default, output is on the standard output stream. You can redirect output to a file by using standard MS-DOS notation, for example:

```
armcc -M source.c > Makefile
```

If you specify multiple source files, a single dependency file is created.

If you specify the -o *filename* option (see *Specifying output format* on page 2-37), the dependency lines generated on standard output refer to filename.o, and not to source.o. However, no object file is produced with the combination of -M -o *filename*.

Use the --md option to generate dependency lines and object files for each source file.

-D*name* [(*parm-list*)] [=*def*]

Defines a macro name as *def*. If =*def* is omitted, the compiler defines *name* to expand as the value 1. You can define function-style macros by appending a macro parameter list to *name*. The compiler defines macro names based on the command-line options. For a list, see *Predefined macros* on page 3-89.

-U*name*   Removes any initial definition of the macro *name*. This has the same effect
                  as the text #undef *name* at the head of the source file. You can repeat this
                  option.

Ordering macro definitions

Where you specify macro options, the compiler defines and undefines macros in the
following order:
1.   compiler predefined macros (see *Predefined macros* on page 3-89).
2.   macros defined explicitly, using -D*name*.
3.   macros explicitly undefined, using -U*name*.

2.3.7   Syntax-checking

This option enables you to perform syntax-checking on your source code:

--no_code_gen

> Instructs the compiler to perform syntax-checking only, without creating
> an object file.

2.3.8   C++ Language configuration and object generation

These options enable you to control various elements of the C++ compilation:

--exceptions --no_exceptions

> Enables or disables C++ exception handling (see *C++ exception
> handling* on page 3-63). The default behavior is --no_exceptions.
>
> Compiling with exceptions enabled causes the compiler to emit
> unwinding tables to support exception propagation at runtime. Also in
> C++ it enables the use of the throw and try/catch function exception
> specifications.
>
> ———— **Note** ————
> When --no_exceptions is in force, throw and try/catch are not permitted
> in source code. However, function exception specifications are still
> parsed, but most of their meaning is ignored.
> ————————————————

--exceptions_unwind --no_exceptions_unwind

> Enables or disables function unwinding for exceptions-aware C++ code
> (see *Function unwinding at runtime* on page 3-63). The default behavior
> is --exceptions_unwind.

You must also specify `--exceptions` if you want to use these options.

`--export_all_vtbl`

Enables you to control how dynamic symbols are exported. Use this option to export all virtual table (vtable) functions and *Runtime Type Information* (RTTI) for classes with a key function. A key function is the first virtual function of a classs, in declaration order, that is not inline, and is not pure virtual.

You can disable export for specific classes by using `__declspec(notshared)`. For example:

```
struct __declspec(notshared) X { virtual int f(); };
                                    // Do not export this
int X::f() {return 1;}
struct Y : X { virtual int g(); }; // Do export this
int Y::g() {return 1;}
```

`--export_defs_implicitly`

Enables you to control how dynamic symbols are exported. Use this option to export definitions where the prototype was marked `dllimport`.

`--dllexport_all`

Enables you to control symbol visibility when building DLLs. Use this option to mark all `extern` definitions as `__declspec(dllexport)`.

`--no_hide_all`

Enables you to control symbol visibility when building SVr4 shared objects. Use this option to mark all `extern` definitions as `__declspec(dllexport)`, and to import all undefined references.

`--force_new_nothrow`

The C++ Standard states that only a no throw `operator new` is permitted to return `NULL` on failure. Any other `operator new` is never permitted to return `NULL` and the default `operator new` throws an exception.

If you use `--force_new_nothrow` then the compiler treats expressions

```
new T(...args...)
```

that use the global `::operator new` or `::operator new[]` as if they are

```
new (std::nothrow) T(...args...)
```

`--force_new_nothrow` also causes any class-specific `operator new` or any overloaded global `operator new` to be treated as `throw()`, for example:

```
struct S {
    void* operator new(std::size_t);
    void* operator new[](std::size_t);
};
void *operator new(std::size_t, int);
```

This is treated as:

```
struct S {
    void* operator new(std::size_t) throw();
    void* operator new[](std::size_t) throw();
};
void *operator new(std::size_t, int) throw();
```

For more details on `operator new`, see *Using the ::operator new function* on page 3-58.

The default is to follow the C++ Standard.

`--implicit_include --no_implicit_include`

Enable or disable implicit inclusion of source files as a method of finding definitions of template entities to be instantiated. See *Template instantiation* on page 3-60. The default behavior is `--implicit_include`.

`--pending_instantiations=`*n*

Specifies the maximum number of concurrent instantiations of a given template that can be in the process of being instantiated. This is used to detect runaway recursive instantiations. If *n* is zero, there is no limit. The default value is 64.

`--nonstd_qualifier_deduction --no_nonstd_qualifier_deduction`

Controls whether or not nonstandard template argument deduction is to be performed in the qualifier portion of a qualified name. With this feature enabled, a template argument for the template parameter T can be deduced in contexts like A<T>::B or T::B. The standard deduction mechanism treats these as non-deduced contexts that use the values of template parameters that were either explicitly specified or deduced elsewhere. The default is `--no_nonstd_qualifier_deduction`.

`--rtti --no_rtti`Enables or disables support for RTTI features `dynamic_cast` and `typeid`. The default behavior is `--rtti`.

`--using_std --no_using_std`

Enables or disables implicit use of the `std` namespace when standard header files are included. See *Namespaces* on page 3-61 for more details. The default behavior is `--no_using_std`.

--old_specializations --no_old_specializations

> Enables or disables the acceptance of old-style template specializations. That is, specializations that do not use the template<> syntax. The default behavior is --no_old_specializations.

--guiding_decls --no_guiding_decls

> Enables or disables the recognition of guiding declarations of template functions. A *guiding declaration* is a function declaration that matches an instance of a function template but has no explicit definition (because its definition derives from the function template), for example:
>
> ```
> template <class T> void f(T) { ... }
> void f(int);
> ```
>
> When regarded as a guiding declaration, f(int) is an instance of the template. Otherwise, it is an independent function so you must supply a definition.
>
> If --no_guiding_decls is combined with --old_specializations, a specialization of a non-member template function is not recognized. It is treated as a definition of an independent function.
>
> The default behavior is --no_guiding_decls.

--implicit_typename --no_implicit_typename

> Enables or disables implicit determination, from context, whether a template parameter dependent name is a type or nontype. The default behavior is --no_implicit_typename.
>
> ———— **Note** ————
>
> The --implicit_typename option has no effect unless you also specifiy --no_parse_templates.
>
> ————————————
>
> See *Template instantiation* on page 3-60, for more details.

--parse_templates --no_parse_templates

> Enables or disables the parsing of non-class templates in their generic form, that is, even if they are not actually instantiated. If dependent name processing is enabled, then parsing is done by default.
>
> See *Template instantiation* on page 3-60, for more details.

--dep_name --no_dep_name

> Enables or disables dependent name processing. That is, the separate lookup of names in templates at the time the template is parsed, and at the time it is instantiated.

The default behavior is `--dep_name`.

―――― **Note** ――――

`--dep_name` cannot be combined with `--no_parse_templates`. Attempting to use these options together, generates the following error:

```
#1054: option "dep_name" cannot be used with "no_parse_templates"
```

See *Template instantiation* on page 3-60, for more details.

`--friend_injection --no_friend_injection`

In C++, these options control whether or not the name of a class or function that is declared only in `friend` declarations is visible when using the normal lookup mechanisms.

When `friend` names are injected, they are visible to these lookups. When `friend` names are not injected (as required by the standard), function names are visible only when using argument-dependent lookup, and class names are never visible.

For details on `friend` declarations, see *friend* on page 3-34.

2.3.9   Specifying output format

By default, source files are compiled and linked into an executable image.

These options enable you to direct the compiler to create unlinked object files, assembly language files, or listing files from C or C++ source files.

-c   Compiles but does not perform the link phase. The compiler compiles the source program and writes the object files to either the current directory or the file specified by the -o *file* option. This option is different from the uppercase -C option described in *Setting preprocessor options* on page 2-31. (The -C option retains comments in preprocessor output.)

--list   Generates raw listing information in a file with a name based on the input filename. Typically, this information is used to generate a formatted listing. The raw listing file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the compiler. Each line of the listing file begins with any of the following key characters that identifies the type of line:

> N   A normal line of source. The rest of the line is the text of the line.

> X   The expanded form of a normal line of source. The rest of the line is the text of the line. This line appears following the N line, and only if the line contains nontrivial modifications. Comments are considered trivial modifications, and macro expansions, line splices, and trigraphs are considered nontrivial modifications. Comments are replaced by a single space in the expanded-form line.

> S   A line of source skipped by an #if or similar. The rest of the line is text.

> > ——— **Note** ———
> >
> > The #else, #elif, or #endif that ends a skip is marked with an N.

> L   An indication of a change in source position. That is, the line has a format similar to the # line-identifying directive output by cpp:
> >
> > L *line-number* "*file-name*" *key*
> > where *key* can be:
> >
> > 1   For entry into an include file.
> >
> > 2   For exit from an include file.

Copyright © 2005 ARM Limited. All rights reserved.

Otherwise, *key* is omitted. The first line in the raw listing file is always an L line identifying the primary input file. L lines are also output for #line directives (where *key* is omitted). L lines indicate the source position of the following source line in the raw listing file.

R/W/E   An indication of a diagnostic, where:

R   Indicates a remark.

W   Indicates a warning.

E   Indicates an error.

The line has the form:

type "*file-name*" *line-number column-number message-text*

where *type* can be R, W, or E. See *Severity of diagnostic messages* on page 2-59 for more details.

Errors at the end of file indicate the last line of the primary source file and a column number of zero.

Command-line errors are errors with a filename of "<command line>". No line or column number is displayed as part of the error message.

Internal errors are errors with position information as usual, and message-text beginning with (Internal fault).

When a diagnostic message displays a list (for example, all the contending routines when there is ambiguity on an overloaded call), the initial diagnostic line is followed by one or more lines with the same overall format (code letter, filename, line number, column number, and message text). However, the code letter is the lowercase version of the code letter in the initial line. The source position in these lines is the same as that in the corresponding initial line.

-o *file*   Names the file that holds the final output of the compilation:

- If *file* is -, the output is written to the standard output stream and -E must be specified. This can also be written as -o-.

- Used with -c, it names the object file.

- Used with -E, it specifies the output file for preprocessed source.

- If none of -c or -E is present, it specifies the output file of the link step. An executable image called *file*.axf is created.

If you do not specify a -o option, the name of the output file defaults to the name of the input file with the appropriate filename extension. For example, the output from file1.c is named file1.o if the -c option is specified. If none of -c, -E, or -o is present the default linker output name of __image.axf is used.

——— **Note** ———

This option overrides the --default_extension option (see *Default object extension* on page 2-57).

--depend *filename* --mdThis option compiles the source and writes make file dependency lines to a file. The output file is suitable for use by a make utility

If you use the --md option, the compiler names the file *filename*.d, where *filename* is the name of the source file. If you specify multiple source files, a dependency file is created for each source file.

If you use the --depend option, you can specify any filename. However, if you specify multiple source files, --depend operates in the same way as the --md option, and ignores the dependency filename that you have specified.

2.3.10 Specifying the target processor or architecture

These options enable you to specify the target processor or architecture attributes for a compilation. The compiler can take advantage of certain extra features of the selected processor or architecture, such as support for halfword load and store instructions and instruction scheduling.

——— **Note** ———

Specifying the target processor might make the code incompatible with other ARM processors.

You can specify how the compiler is configured at start-up using either:

--arm   To target the ARM instruction set. This is the default.

--thumb   To target the Thumb instruction set.

For more details on these options, see *Targeting the instruction set* on page 2-43.

The following general points apply to processor and architecture options:

•   The supported --cpu values are all current ARM product names or architecture versions. There are no aliases or wildcard matching.

- If you specify an architecture name for the `--cpu` option, the code is compiled to run on any processor supporting that architecture. For example, `--cpu 5TE` produces code that can be used by either the ARM926EJ-S™ or ARM946E-S™.

- If you specify a processor for the `--cpu` option, for example `--cpu ARM926EJ-S`, the compiled code is optimized for that processor. This enables the compiler to use specific coprocessors or instruction scheduling for optimum performance.

- Use only a single processor or architecture name with `--cpu`. You cannot specify both a processor and an architecture.

- If `--cpu` is not specified, the default is `--cpu 5TE`.

- Specifying a processor that supports Thumb instructions, such as `--cpu ARM926EJ-S` does not make the compiler generate Thumb code. It only enables features of the processor to be used, such as long multiply. Use the `--thumb` option to generate Thumb code.

The following options are available:

`--cpu list` Lists the supported architecture and processor names that you can use with the `--cpu` *name* option.

`--cpu` *name* This option generates code for a specific ARM processor or architecture.

If *name* is a processor:

- You must enter the name exactly as it is shown on ARM data sheets, for example `ARM926EJ-S`. Wildcard characters are not accepted.

- Selecting the processor selects the appropriate architecture, *Floating-Point Unit* (FPU), and memory organization.

- Some specifications of `--cpu` imply an `--fpu` selection. For example, when compiling with the `--arm` option, `--cpu ARM1136JF-S` implies `--fpu vfpv2`. Be aware that any *explicit* FPU (set with `--fpu`) on the command line overrides an *implicit* FPU.

- If no `--fpu` option and no `--cpu` option are specified, `--fpu softvfp` is used.

If *name* is an architecture, it must be one of:

5TE ARMv5 with long multiply, Thumb, interworking, DSP multiply, and double-word instructions.

`--fpu list` Lists the supported FPU architecture names that you can use with the `--fpu` *name* option.

Deprecated options are not listed.

--fpu *name*   This option determines the target FPU architecture. If you specify this option, it overrides any implicit FPU option that appears on the command line, for example, where you use the --cpu option.

Valid options for *name* are:

none   Selects no floating-point option. No floating-point code is to be used. This produces an error if your code contains floats.

softvfp   Selects the software floating-point library (fplib). This is the default if you do not specify a --fpu option, or if you select a CPU that does not have an FPU.

To get a full list of FPU architectures use the --fpu list option.

2.3.11   Generating debug information

These options enable you to specify whether debug tables are generated for the current compilation. If debug tables are generated, these options also enable you to specify the format of the debug tables. See *Pragmas* on page 3-2 for more information on controlling debug information.

———— **Note** ————

Optimization criteria can limit the debug information generated by the compiler. See *Defining optimization criteria* on page 2-43 for more information.

Compilation tools are restricted to the permitted endianness of the target microcontroller unit (MCU).

Debug table generation options

The following options specify how debug tables are generated:

-g --debug   This option switches on the generation of debug tables for the current compilation. The compiler produces the same code whether or not -g is used. The only difference is the existence of debug tables. --debug is a synonym for -g.

Using -g does not affect optimization settings. These options are specified by -0 (for more details, see *Multi-optimization options* on page 2-43). By default, using the -g option alone is equivalent to:

```
-g -dwarf2 --debug_macros
```

———— **Note** ————

Do not rely on the implementation details of this default behavior, because it might change in future releases. The compiler issues a warning if you use the -g option without an optimization option.

--no_debug   This option switches off the generation of debug tables for the current compilation. This is the default option.

--no_debug_macros

This option, when used with -g, switches off the generation of debug table entries for preprocessor macro definitions. This can reduce the size of the debug image.

--debug_macros

This option, when used with -g, enables the generation of debug table entries for preprocessor macro definitions. This is the default. If your debugger ignores preprocessor entries, you can use --no_debug_macros to reduce the size of the debug image.

Debug table format options

The following options specify the format of debug tables generated by the compiler:

--dwarf2   This option specifies DWARF2 debug table format. This is the default.

--dwarf3   This option specifies DWARF3 debug table format.

2.3.12   Controlling code generation

Use the options described in this section to control aspects of the code generated by the compiler such as optimization. See *Pragmas* on page 3-2 for information on additional code generation options that are controlled using pragmas.

This section describes:
- *Targeting the instruction set* on page 2-43
- *Setting byte order* on page 2-43
- *Defining optimization criteria* on page 2-43
- *Controlling code and data sections* on page 2-51
- *Setting pointer alignment options* on page 2-52
- *Setting alignment options* on page 2-53
- *Controlling implementation details* on page 2-54.

Targeting the instruction set

These options control the target instruction set:

--arm   Configures the compiler to target the ARM instruction set. This is the
                default.

--thumb   Configures the compiler to target the Thumb instruction set. This
                predefines `__thumb` and `__thumb__`.

Also, see the descriptions of `#pragma arm` and `#pragma thumb` in *Pragmas controlling code generation* on page 3-5. These pragmas enable you to compile specific functions for ARM or Thumb.

If you are compiling code that is targeted at both ARM and Thumb, then you must specify the interworking option --apcs /interwork. See *Interworking qualifiers* on page 2-65 for more details.

If you enter `armcc --thumb --fpu vfp` on the command line, the compiler compiles as much of the code using the Thumb instruction set as possible. However, the compiler might generate ARM code for some parts of the compilation. See details on the argument `--fpu name` in *Specifying the target processor or architecture* on page 2-39.

Setting byte order

These options control endianness:

--littleend   This option generates code for an ARM processor using little-endian
                memory. With little-endian memory, the least significant byte of a word
                has the lowest address. This is the default.

Defining optimization criteria

The optimization options can be grouped into:
*   *Multi-optimization options*
*   *Single-optimization options* on page 2-48.

Multi-optimization options

The optimizations described in this section enable you to control multiple optimizations with a single option.

You can also apply the -O*num*, -Ospace, and -Otime optimizations on individual functions using pragmas. See *Pragmas controlling multiple optimizations* on page 3-5 for more information.

The optimization options that are prefixed by -O can be specified using lowercase, uppercase, or mixed-case. However, the -O prefix must be uppercase. For example:

```
-Ospace
-OTIME
-OSpace
```

The multi-optimization options are:

-O, *num*   Specifies the level of optimization to be used. The optimization levels are:

    -O0   Minimum optimization. Turns off most optimizations. It gives the best possible debug view and the lowest level of optimization.

    -O1   Restricted optimization. Removes unused inline functions and unused static functions. Turns off optimizations that seriously degrade the debug view. If used with --debug (see *Debug table generation options* on page 2-41), this option gives a satisfactory debug view with good code density.

    -O2   High optimization. If used with --debug (see *Debug table generation options* on page 2-41), the debug view might be less satisfactory because the mapping of object code to source code is not always clear.

        This is the default optimization level.

    -O3   Maximum optimization. The balance between space and time optimizations in the generated code is more heavily weighted towards space or time compared with -O2. That is:

- -O3 -Otime aims to produce faster code than -O2 -Otime, at the risk of increasing your image size

- -O3 -Ospace aims to produce smaller code than -O2 -Ospace, but performance might be degraded.

        -O3 performs the same optimizations as -O2. In addition, -O3 performs extra optimizations that are more aggressive, such as:

- More aggressive inlining and automatic inlining for -O3 -Otime.

- Multifile compilation by default.

        For more details on multifile compilation, see *Multifile compilation* on page 2-6. Also, see the description of the --multifile option.

——— **Note** ———

For floating-point code, -03 is not necessarily ISO C and C++ standard-compliant. Use `-03 --fpmode=std` to ensure ISO compliance. See the description of `--fpmode` for more information.

——— **Note** ———

Do not rely on the implementation details of these optimizations, because they might change in future releases.

-Ospace   Instructs the compiler to perform optimizations to reduce image size at the expense of a possible increase in execution time. For example, large structure copies are done by out-of-line function calls instead of inline code. Use this option if code size is more critical than performance. This is the default.

-Otime   Instructs the compiler to perform optimizations to reduce execution time at the possible expense of a larger image. Use this option if execution time is more critical than code size. For example, it compiles:

```
while (expression) body;
```

as:

```
if (expression) {
    do body;
    while (expression);
}
```

If you specify neither -Otime or -Ospace, the compiler uses -Ospace. You can compile time-critical parts of your code with -Otime, and the rest with -Ospace.

You must not specify both -Otime and -Ospace in the same compiler invocation. If you do, the last one wins (see *Ordering command-line options* on page 2-21).

--feedback *filename*

Specifies the feedback file created by a previous execution of the ARM linker. The file contains a list of functions that the linker identifies as being unused in your code. The contents of this file are optimization hints only. These hints might be ignored by the compiler. Therefore, this is a safe optimization.

See *Linker feedback* on page 2-7 for more details.

──────── **Note** ────────

It is recommended that you use this optimization in preference to the `--split_sections` option (formerly `-zo`) for removing unused functions. This is because linker feedback produces smaller code, by avoiding the overhead of splitting all sections.

────────────────────

`--fpmode` *mode1*

Specifies the floating-point conformance, and sets library attributes and floating-point optimizations. *mode1* can be one of:

`ieee_full` All facilities, operations, and representations guaranteed by the IEEE standard are available in single and double-precision. Modes of operation can be selected dynamically at runtime.

This defines the symbols:

```
__FP_IEEE
__FP_FENV_EXCEPTIONS
__FP_FENV_ROUNDING
__FP_INEXACT_EXCEPTION
```

`ieee_fixed`

IEEE standard with round-to-nearest and no inexact exception.

This defines the symbols:

```
__FP_IEEE
__FP_FENV_EXCEPTIONS
```

`ieee_no_fenv`

IEEE standard with round-to-nearest and no exceptions. This mode is compatible with the Java floating-point arithmetic model.

This defines the symbol `__FP_IEEE`.

`std` IEEE finite values with denormals flushed to zero, round-to-nearest, and no exceptions. It is C and C++ compatible. This is the default option.

Finite values are as predicted by the IEEE standard. However:

- NaNs and infinities might not be produced in all circumstances defined by the IEEE model. Also, when they are produced, they might not have the same sign.

- The sign of zero might not be that predicted by the IEEE model.

fast   Perform more aggressive floating-point optimizations that might cause a small loss of accuracy to provide a significant performance increase. This option results in behavior that is not fully ISO C and C++ standard-compliant, however numerically robust floating-point programs will behave correctly.

The following optimizations are performed:

- Math functions with **float** arguments, or **double** arguments that have been converted from **float**, call the single precision version if it exists, unless the argument is a constant expression. This is not compliant with ISO C or C++.

- Double precision floating-point expressions that are narrowed to single precision are evaluated in single precision when it is beneficial to do so. For example, **float** y = (**float**)(x + 1.0) is evaluated as **float** y = (**float**)x + 1.0f.

- Division by a floating-point constant is replaced by multiplication with the inverse. For example, x / 3.0 is evaluated as x * (1.0 / 3.0).

This option defines the symbol __FP_FAST.

--multifile   Enables the compiler to perform optimization across all specified files, instead of on each individual file. The specified files are compiled into one single object file. Using --multifile requires large amounts of memory while compiling. Although there is no limit to the number of files you can specify on the command line, a practical limit is 10 source files. For more details on multifile compilation, see *Multifile compilation* on page 2-6.

------- **Note** -------

This optimization is on by default for optimization level -03.

--vfe --no_vfe Enables or disables unused virtual function elimination (VFE) in C++ mode. --vfe is the default, except for the case where legacy object files compiled with a pre-RVCT v2.1 compiler do not contain VFE information.

When VFE is enabled, the compiler places the information in special sections with the prefix .arm_vfe_. These sections are harmless to a linker that is not VFE-aware, because they are not referenced by the rest of the

---

code. Therefore, they do not increase the size of the executable. However, they increase the size of the object files. If this is a problem, then specify `--no_vfe`.

For more details on VFE, and the associated linker options, see *RealView Developer Kit v2.2 Linker and Utilities Guide*. Also, see *Calling a pure virtual function* on page C-3 for more information on pure virtual functions.

Single-optimization options

These options enable you to have individual control of the compiler optimizations:

`--autoinline --no_autoinline`

Enables or disables automatic inlining. `--no_autoinline` is the default for optimization levels `-O0` and `-O1`, and `--autoinline` is the default for optimization levels `-O2` and `-O3` (see *Multi-optimization options* on page 2-43).

The compiler automatically inlines functions where it is sensible to do so. The `-Ospace` and `-Otime` options influence how the compiler automatically inlines functions. Selecting `-Otime` increases the likelihood that functions are inlined.

`--data_reorder --no_data_reorder`

Enables or disables automatic reordering of top-level data items (globals, for example). The compiler can save memory by eliminating wasted space between data items. However, `--data_reorder` can break legacy code, if the code makes invalid assumptions about ordering of data by the compiler.

The ISO C Standard does not guarantee data order, so you must avoid writing code that depends on any assumed ordering. If you require data ordering, place the data items into a structure.

`--forceinline`

If this option is used, the compiler always attempts to inline those functions marked as `__inline`, if possible. The compiler attempts to inline the function, regardless of the characteristics of the function. However, the compiler does not inline a function if doing so causes problems, for example, a recursive function is inlined only once.

If you want to force specific functions to be inlined, use the `__forceinline` function storage class modifier (see *Function storage class modifiers* on page 3-13).

--no_inline   Disables inlining of functions (see --inline). Calls to inline functions are not expanded inline. You can use this option to help debug inline functions.

If a function is declared inline, then it is compiled out-of-line into a common code section.

--inline   Enables the compiler to inline functions. This is the default.

The compiler inlines functions as follows:

- Automatically, for optimization levels -02 and -03 (see *Multi-optimization options* on page 2-43), unless you use the option --no_autoinline.

- When the function is qualified as an inline function. That is with the __inline keyword in C, the __forceinline keyword in C and C++, or the inline keyword in C++. This applies for all optimization levels. Functions that are explicitly qualified as inline functions are more likely to be inlined. However using the inline qualifier does not guarantee that functions are inlined. See *Function keywords* on page 3-9. Also, see the description of --forceinline.

The compiler changes the criteria for inlining functions depending on whether you select -Ospace or -Otime. Select -Otime to increase the likelihood that a function is inlined. See *Multi-optimization options* on page 2-43 for more details.

Sometimes, an out-of-line copy of an inlined function might remain in an object or image, even though that code is no longer used. Linker feedback enables you to detect and remove any unused code fragments. See *Linker feedback* on page 2-7.

——— **Note** ———

When you set a breakpoint on an inline function, an ARM debugger attempts to set a breakpoint on each inlined instance of that function. If you are using RVI-ME or other hardware to debug an image in ROM, and the number of inline instances is greater than the number of available hardware breakpoints, the debugger cannot set the additional breakpoints and reports an error.

———————

--lower_ropi --no_lower_ropi

Enables or disables less restrictive C in ROPI mode. See *Position independence qualifiers* on page 2-65 for details of the /ropi option.

——— **Note** ———

If you compile with `--lower_ropi`, then the static initialization is done at runtime by the C++ constructor mechanism, even for C. This enables these static initializations to work with ROPI code.

`--lower_rwpi --no_lower_rwpi`

Enables or disables less restrictive C and C++ in RWPI mode. `--lower_rwpi` is the default. See *Position independence qualifiers* on page 2-65 for details of the `/rwpi` option.

——— **Note** ———

If you compile with `--lower_rwpi`, then the static initialization is done at runtime by the C++ constructor mechanism, even for C. This enables these static initializations to work with RWPI code.

`--split_ldm`   Instructs the compiler to split `LDM` and `STM` instructions into two or more `LDM` or `STM` instructions, where required, to reduce the maximum number of registers transferred to:

- five, for all `STM`s, and for `LDM`s that do not load the PC
- four, for `LDM`s that load the PC.

The `--split_ldm` option has the following effects:

- It can reduce interrupt latency on ARM systems that:
  — do not have a cache or a write buffer
  — use zero-wait-state, 32-bit memory.

  ——— **Note** ———

  Using `--split_ldm` increases code size and decreases performance slightly.

- It does not split ARM inline assembly `LDM` or `STM` instructions, or VFP `FLDM` or `FSTM` instructions.

There are some systems that do not benefit from being built with `--split_ldm`:

- It has no significant benefit for cached systems, or for processors with a write buffer.

- It has no benefit for systems with non-zero-wait-state memory, or for systems with slow peripheral devices. Interrupt latency in such systems is determined by the number of cycles required for the slowest memory or peripheral access. Typically, this is much greater than the latency introduced by multiple register transfers.

Controlling code and data sections

This option enables you to control code and data sections:

--split_sections

This option generates one ELF section for each function in the source file. Output sections are named with the same name as the function that generates the section, but with an i. prefix. For example:

```
int f(int x) { return x+1; }
```

compiled with --split_sections gives:

```
        AREA ||i.f||, CODE, READONLY
f PROC
        ADD     r0,r0,#1
        MOV     pc,lr
```

This option increases code size slightly (typically by a few percent) for some functions because it reduces the potential for sharing addresses, data, and string literals between functions.

———— **Note** ————

- If you want to remove unused functions, it is recommended that you use the linker feedback optimization in preference to this option. This is because linker feedback produces smaller code, by avoiding the overhead of splitting all sections. See *Linker feedback* on page 2-7 for more details.

- The pragma arm section pragma specifies the code or data section name used for subsequent functions or objects. This includes definitions of anonymous objects that the compiler creates for initializations. See *Pragmas controlling code generation* on page 3-5 for more details.

- Use a scatter-loading description file to place some functions in fast memory and others in slow memory (see the chapter on using scatter-loading description files in *RealView Developer Kit v2.2 Linker and Utilities Guide*).

You can also use a scatter-loading file to place a function at a particular address in memory.

Setting pointer alignment options

This option enables you to control pointer alignment:

`--pointer_alignment=`*num*

Specifies the unaligned pointer support required, where *num* is one of the following:

1   Treats accesses through pointers as having an alignment of one, that is, byte-aligned or unaligned.

2   Treats accesses through pointers as having an alignment of at most two, that is, at most halfword aligned.

4   Treats accesses through pointers as having an alignment of at most four, that is, at most word aligned.

8   Accesses through pointers have normal alignment, that is, at most doubleword aligned.

De-aligning pointers might increase the code size, even on CPUs with unaligned access support. For example, on ARMv6, using the `UL41` memory access model. This is because only a subset of the load and store instructions benefit from unaligned access support. The compiler is unable to use multiple-word transfers or coprocessor-memory transfers, including hardware floating-point loads and stores, directly on unaligned memory objects.

——— **Note** ———

• Code size might increase significantly when compiling for CPUs without hardware support for unaligned access.

• Unaligned pointer mode does not affect the placement of objects in memory, nor the layout and padding of structures.

This option assists the porting of source code that has been written for architectures without alignment requirements. You can achieve finer control of access to unaligned data, with less impact on the quality of generated code, using the `__packed` qualifier. For more details on the `__packed` qualifier, see *Type qualifiers* on page 3-26.

Setting alignment options

These options enable you to control memory alignment:

--memaccess *option*

> This option tells the compiler that the memory in the target system has slightly restricted or expanded capabilities. If you specify a processor that supports ARMv6 (for example, --cpu ARM1136J-S) or the ARMv6 architecture (that is, --cpu 6), the compiler utilizes ARMv6 unaligned access support to speed up accesses to packed structures. See *ARMv6 unaligned accesses* on page 2-54 for more details.
>
> Specify *option* to indicate the load and store capability:
>
> -UL41   Disables unaligned mode for code that uses pre-ARMv6 unaligned access behavior.
>
> It is possible that:
>
> • the processor has memory access modes available that the physical memory lacks (load aligned halfword, for example)
>
> • the physical memory has access modes that the processor cannot use (ARMv3 load aligned halfword, for example).

--min_array_alignment=*option*

> Specifies the minimum alignment of arrays, where *option* is one of the following:
>
> 1   byte alignment, or unaligned
>
> 2   two-byte (halfword) alignment
>
> 4   four-byte (word) alignment
>
> 8   eight-byte (doubleword) alignment.
>
> For example, compiling the following code with --min_array_alignment=8, gives the alignment described in the comments:

```
char arr_c1[1];      // alignment == 8
char c1;             // alignment == 1
char arr_c2[3];      // alignment == 8
char arr_c3[10];     // alignment == 8
struct st {
    int i1;
} c;                 // alignment == 1
char c2;             // alignment == 1
```

> Also, see *Keywords specific to the ARM compiler* on page 3-22 for a description of the __align(*n*) storage class modifier.

ARMv6 unaligned accesses

The compiler utilizes ARMv6 unaligned access support by default. This speeds up accesses to packed structures by enabling an LDR instruction to load from, or an STR instruction to store to, a non-word aligned address. That is, the compiler might generate unaligned word and halfword accesses, and might select a library that supports unaligned accesses. Structures remain unpacked, unless you explicitly qualify them with __packed. For more details on the __packed qualifier, see *Type qualifiers* on page 3-26.

Therefore, code compiled for ARMv6 can run correctly only if you enable unaligned support on the ARM core. To enable unaligned support on an ARMv6 core, you must set the U bit (bit 22) of CP15 register 1 in your initialization code. This can also be achieved in hardware, by tying the **UBITINIT** input to the core HIGH.

To generate code that uses the pre-ARMv6 unaligned access behavior, use the --memaccess -UL41 compiler option.

Controlling implementation details

These options enable you to specify implementation details:

--enum_is_int

> This option forces all enumerations to be stored in integers. This option is switched off by default and the smallest data type is used that can hold the values of all enumerators.
>
> ———— **Note** ————
> The --enum_is_int option is not recommended for general use and is not required for ISO-compatible source. Code compiled with this option is not compliant with the *ABI for the ARM Architecture (base standard)* [BSABI], and incorrect use might result in a failure at runtime. This option is not supported by the C++ libraries.

--dollar --no_dollar Accepts dollar signs, $, in identifiers. The default is --dollar, except in --strict mode.

--alternative_tokens --no_alternative_tokens

> Enables or disables the recognition of alternative tokens. This controls recognition of the digraphs in C and C++, and controls recognition of the operator keywords, such as **and** and **bitand**, in C++. For more details on digraphs, see *The Design and Evolution of C++*, or any other book describing the C++ programming language. The default behavior is --alternative_tokens.

--multibyte_chars --no_multibyte_chars

>Enables or disables processing for multibyte character sequences in comments, string literals, and character constants. Multibyte encodings are used for character sets such as the Japanese *Shift-Japanese Industrial Standard* (Shift-JIS). The default behavior is --multibyte_chars.

--locale *string*

>Use this option in combination with --multibyte_chars to switch the default locale for source files to the one you specify in *string*. For example, to compile Japanese source files on an English-based Windows NT workstation, use:

>--multibyte_chars --locale japanese

>The permitted settings of locale are determined by the host platform.

--loose_implicit_cast

>Makes illegal implicit casts legal, such as implicit casts of a nonzero **int** to **pointer**, for example:

>int *p = 0x8000;

>Without this option, the compiler reports:

>Error: #144: a value of type "int" cannot be used to initialize an entity of type "int *"

>With this option, the compiler generates the following warning message, which you can suppress (see *Suppressing diagnostic messages* on page 2-62):

>Warning: #152-D: conversion of nonzero integer to pointer

--restrict --no_restrict

>Enables or disables the use of the **restrict** keyword. The default is --no_restrict.

>See *restrict* on page 3-34 for more details on the **restrict** keyword.

--signed_chars --unsigned_chars

>Makes the **char** type to be signed or unsigned. The default is --unsigned_chars.

>When **char** is signed, the macro __FEATURE_SIGNED_CHAR is defined by the compiler.

For --unsigned_chars, any **char** that is assigned a negative number causes the following warning to be generated:

```
Warning:  #68-D: integer conversion resulted in a change of sign
```

───── **Note** ─────

The --signed_chars option is not recommended for general use and is not required for ISO-compatible source. Code compiled with this option is not compliant with the *ABI for the ARM Architecture (base standard)* [BSABI], and incorrect use might result in a failure at runtime. This option is not supported by the C++ libraries.

2.3.13   Adding symbol versions

The ARM linker conforms to the *Base Platform ABI for the ARM Architecture* [BPABI] (unpublished DRAFT) and supports the GNU-extended symbol versioning model.

To create a function with a symbol version in C or C++ code, you must use the assembler label GNU extension to rename the function symbol into a symbol that has the name *function@@ver* for a default *ver* of *function*, or *function@ver* for a non default *ver* of *function* (see *Assembler labels* on page 3-68).

For example, to define a default version:

```
int new_function(void) __asm__("versioned_fun@@ver2");
int new_function(void)
{
    return 2;
}
```

To define a non default version:

```
int old_function(void) __asm__("versioned_fun@ver1");
int old_function(void)
{
    return 1;
}
```

See the chapter describing how to access symbols in *RealView Developer Kit v2.2 Linker and Utilities Guide* for a full description of symbol versioning in RVDK.

───── **Note** ─────

Compilation tools are restricted to the permitted endianness of the target MCU.

───────

2.3.14   Default object extension

This option enables you to change the default extension for object files:

`--default_extension` *ext*

Enables you to change the extension for object files from the default extension (`.o`) to that specified by *ext*. The following example creates an object file called `test.obj`, instead of `test.o`:

```
armcc --default_extension obj -c test.c
```

─────── **Note** ───────

The `-o` *filename* option overrides this (see *Specifying output format* on page 2-37). For example, the following command still gives the specified object file the `.o` extension:

```
armcc --default_extension obj -o test.o -c test.c
```

─────────────────────

2.3.15   Diagnostic messages

The ARM compiler issues messages about potential portability problems and other hazards. This section describes compiler options that you can use to:

• Turn off specific messages. For example, you can turn off warnings if you are in the early stages of porting a program written in old-style C. In general, however, it is better to check the code than to switch off messages.

• Change the severity of specific messages.

——— **Note** ———

The previous release of RVCT supported old compiler options to help you to migrate your message options to the new release. However, these options are now obsolete and are no longer supported. See *Mapping old compiler options to the new release* on page E-2 for details.

This section describes:
• *Severity of diagnostic messages* on page 2-59
• *Controlling the output of diagnostic messages* on page 2-60
• *Changing the severity of diagnostic messages* on page 2-61
• *Suppressing diagnostic messages* on page 2-62
• *Suppressing warning messages with the -W option* on page 2-62
• *Exit status codes and termination messages* on page 2-62.

Severity of diagnostic messages

Diagnostic messages have an associated *severity*, as described in Table 2-6.

Table 2-6 Severity of diagnostic messages

Severity	Description
Internal error	Internal errors indicate an internal problem with the compiler. Contact your supplier with the details listed in *Feedback on RealView Developer Kit* on page xiii.
Error	Errors indicate problems that cause the compilation to stop. These errors include command line errors, internal errors, missing include files, and violations in the syntactic or semantic rules of the C or C++ language. If multiple source files are specified, then no further source files are compiled. For example: `Error: #65: expected a ";"`
Warning	Warnings indicate unusual conditions in your code that might indicate a problem. Compilation continues, and object code is generated unless any further problems with an Error severity are detected. For example: `Warning: #1293-D: assignment in condition`
Remark	Remarks indicate common, but sometimes unconventional, use of C or C++. These diagnostics are not displayed by default. Compilation continues, and object code is generated unless any further problems with an Error severity are detected. For example: `#940-D: missing return statement at end of non-void function "main"`

Controlling the output of diagnostic messages

These options enable you to control the output of diagnostic messages:

--brief_diagnostics --no_brief_diagnostics

> Enables or disables a mode where a shorter form of the diagnostic output is used. When enabled, the original source line is not displayed and the error message text is not wrapped when it is too long to fit on a single line. The default is --no_brief_diagnostics.

--diag_style {arm|ide}

> Specifies the style used to display diagnostic messages:

> arm   Display messages using the ARM compiler style. This is the default if --diag_style is not specified. For example:

> > ```
> > "test.c", line 352: Warning:  #177-D: variable "temp" was
> > declared but never referenced
> > ```

> ide   Include the line number and character count for the line that is in error. These values are displayed in parentheses. For example:

> > ```
> > test.c(352,10) : Warning  #177-D: variable "temp" was
> > declared but never referenced
> > ```

--errors *efile*

> Redirects the output of diagnostic messages from stderr to the specified file *efile*. This option is useful on systems where output redirection of files is not well supported.

> Diagnostics that relate to problems with the command options are not redirected. For example, if you type an option name incorrectly. However, if you specify an incorrect argument to an option (for example, --cpu 999), the related diagnostic is redirected to the specified *efile*.

--remarks   Causes the compiler to issue remark messages. Remarks are not issued by default.

--wrap_diagnostics --no_wrap_diagnostics

> Enables or disables the wrapping of error message text when it is too long to fit on a single line.

Changing the severity of diagnostic messages

These options enable you to change the diagnostic severity of all remarks and warnings, and a limited number of errors:

`--diag_error` *tag*[, *tag*, ...]

>   Sets the diagnostic messages that have the specified tag(s) to Error severity.

`--diag_remark` *tag*[, *tag*, ...]

>   Sets the diagnostic messages that have the specified tag(s) to Remark severity.

`--diag_warning` *tag*[, *tag*, ...]

>   Sets the diagnostic messages that have the specified tag(s) to Warning severity.

These options require a comma-separated list of error message numbers that specifies the messages that must be changed. For example, you might want to change the following warning message to Remark severity, because remarks are not displayed by default:

```
Warning:  #1293-D: assignment in condition - give arg types
```

To do this, use the following command:

```
armcc --diag_remark 1293 ...
```

――――― **Note** ―――――

These options also have pragma equivalents. See *Pragmas controlling diagnostic messages* on page 3-8 for details.

―――――――――――――――――

The following diagnostic messages can be changed:

*   Messages with the number format *#nnnn*-D, for example:

    ```
    Warning:  #1293-D: assignment in condition - give arg types
    ```

*   Warning messages with the number format C*nnnn*W, for example:

    ```
    Warning: C2874W: y may be used before being set
    ```

Suppressing diagnostic messages

This option enables you to suppress diagnostic messages:

`--diag_suppress` *tag*[, *tag*, ...]

This option disables all diagnostic messages that have the specified tag(s).

This option requires a comma-separated list of diagnostic message numbers that specifies the messages that must be suppressed. For example, to suppress the warning messages that have numbers 1293 and 187, use the following command:

```
armcc --diag_suppress 1293,187 ...
```

In some circumstances, the compiler issues old-style warning messages, as well as new-style messages. For example, the compiler might issue the following warnings when compiling inline assembly code:

```
Warning:  #1287-D: LDM/STM instruction may be expanded
Warning: C2874W: y may be used before being set
```

To suppress both of these messages, specify only the numerical part of the warning code, for example:

```
armcc --diag_suppress 1287,2874 ...
```

———— **Note** ————

This option also has a pragma equivalent. See *Pragmas controlling diagnostic messages* on page 3-8 for details.

Suppressing warning messages with the -W option

The `-W` option suppresses all warnings.

Exit status codes and termination messages

If the compiler detects any warnings or errors during compilation, the compiler writes the messages to `stderr`. At the end of the messages, a summary message is displayed that gives the total number of each type of message:

filename: *n* `warnings,` *n* `errors`

n indicates the number of warnings or errors detected.

Note

Remarks are not displayed by default. To display remarks, use the `--remarks` compiler option. No summary message is displayed if only remark messages are generated.

Response to signals

The signals **SIGINT** (caused by a user interrupt, like ^C) and **SIGTERM** (caused by a UNIX `kill` command) are trapped by the compiler and cause abnormal termination.

Exit status

On completion, the compiler returns a value greater than zero if an error is detected. See *Severity of diagnostic messages* on page 2-59 for details on how the compiler handles the different levels of diagnostic messages.

Data flow warnings

The compiler issues warnings about the use of uninitialized variables based on its dataflow analysis. The compiler does more of this analysis at higher optimization and so more warnings might be produced at higher optimization levels. For example:

```
int f() { int i; return i++; }
```

produces the following warning at -02:

```
Warning: C2874W: i may be used before being set
```

The check is pessimistic and sometimes reports an anomaly where there is none, for example, when building the Dhrystone example you might see:

```
"dhry_1.c", line 337: Warning: C2874W: Enum_Loc may be used before being set
"dhry_1.c", line 89: Warning: C2874W: Int_2_Loc may be used before being set
```

Note

Data flow warnings are issued by default. In previous versions of the compiler, data flow warnings were issued only if the `-fa` option was specified.

2.3.16   Using AAPCS options

The *Procedure Call Standard for the ARM Architecture* (AAPCS) forms part of the *Application Binary Interface (ABI) for the ARM Architecture (base standard)* [BSABI] specification. By writing code that adheres to the AAPCS, you can ensure that separately compiled and assembled modules can work together.

By specifying qualifiers to the --apcs command-line option, you can define the variant of the AAPCS standard used by the compiler. Here:

• at least one qualifier must be present
• there must be a space between --apcs and the list of qualifiers
• there must be no space between the qualifiers.

If you do not specify --apcs or --cpu, the default is:

```
--apcs /noswst/nointer/noropi/norwpi --cpu 5TE --fpu softvfp
```

However, the default --fpu might be overridden by the use of --cpu. See *Specifying the target processor or architecture* on page 2-39 for more information.

The qualifiers are described in:

• *ADS ABI qualifier*
• *Interworking qualifiers* on page 2-65
• *Position independence qualifiers* on page 2-65
• *Stack checking qualifiers* on page 2-68.

Also see *Controlling code generation* on page 2-42 for other build options, and *Defining optimization criteria* on page 2-43.

ADS ABI qualifier

The compiler supports the option --apcs /adsabi to compile code that is compatible with the old ADS *Application Binary Interface* (ABI). However, this is deprecated and will be removed in a future release:

/adsabi   Generates code that is compatible with the older ADS ABI.

In RVCT v2.x, **double** and **long long** data types are eight-byte aligned, which enables efficient use of the LDRD and STRD instructions in ARMv5TE and later. The /adsabi option changes the alignment back to four bytes. Also out-of-line __inline functions use the old area naming scheme.

Use this option to enable:

• RVCT v2.x C objects to be used with legacy ADS C or C++ objects

- RVCT v2.x C++ objects to be used with legacy ADS C objects.

    ———— **Note** ————

    This only works if your C++ code does not use any Rogue Wave Standard C++ libraries, because these libraries not compatible with objects compiled with `--apcs /adsabi`.

———— **Note** ————

This option will be removed in the future. ARM recommends that you rebuild any legacy ADS objects with RVCT v2.2.

Interworking qualifiers

These `--apcs` qualifiers control interworking:

/nointerwork Generates code with no ARM/Thumb interworking support. This is the default unless you specify a `--cpu` option that corresponds to architecture ARMv5T or later.

/interwork Generates code with ARM/Thumb interworking support.

This is the default for ARMv5T or later, because ARMv5T or later provides direct interworking support. For example, you can specify the processor `--cpu ARM1020E`, which implements architecture ARMv5TE, or you can specify the architecture with `--cpu 5TE`.

See the chapter on using the basic linker functionality in *RealView Developer Kit v2.2 Linker and Utilities Guide* for information on the automatically generated interworking veneers.

Position independence qualifiers

These `--apcs` qualifiers control position independence. These qualifiers also affect the creation of reentrant and thread-safe code, see *Writing reentrant and thread-safe code* on page 5-6 for more details.

/noropi Generates code that is not read-only position-independent. This is the default. `/nopic` is an alias for this option.

/ropi Generates (read-only) position-independent code. `/pic`, for position-independent code, is an alias for this option.

If you select this option, the compiler:

- addresses read-only code and data pc-relative

•   sets the *Position Independent* (PI) attribute on read-only output sections.

There are restrictions when using this qualifier. See *Restrictions on position independent code and data* on page 2-67 for details.

Also see the --lower_ropi option in *Defining optimization criteria* on page 2-43.

———— **Note** ————

--apcs /ropi is not supported when compiling C++. You can compile only the C subset of C++ with /ropi.

/norwpi   Generates code that does not address read/write data position-independently. This is the default. /nopid is an alias for this option.

/rwpi   Generates code that addresses read/write data position-independently, *Read-Write Position Independent* (RWPI). /pid, for position-independent data, is an alias for this option.

If you select this option, the compiler:

•   Addresses writable data using offsets from the static base register sb. This means that:
       —   data address can be fixed at runtime
       —   data can have multiple instances
       —   data can be, but does not have to be, position-independent.
•   Sets the PI attribute on read/write output sections.

There are restrictions when using this qualifier. See *Restrictions on position independent code and data* on page 2-67 for details.

———— **Note** ————

Because the --lower_rwpi option is the default (see *Defining optimization criteria* on page 2-43), code that is not RWPI is automatically transformed into equivalent code that is RWPI. This static initialization is done at runtime by the C++ constructor mechanism, even for C.

/fpic   Generates read-only position-independent code where relative address references are independent of the location where your program is loaded. Relative addressing is only implemented when your code makes use of System V shared libraries. If your code uses shared objects, you do not have to compile with /fpic.

If you select this option, the compiler:

- accesses all static data using PC-relative addressing

- accesses all imported or exported read-write data using a *global offset table* (GOT) entry created by the linker

- accesses all read-only data relative to the PC.

There are restrictions when using this qualifier. See *Restrictions on position independent code and data* for details.

Restrictions on position independent code and data

There are restrictions when you compile code with /ropi, or /rwpi, or /fpic. The main restrictions are:

- The use of --apcs /ropi is not supported when compiling C++. You can compile only the C subset of C++ with /ropi.

- Some constructs that are legal C do not work when compiled for --apcs /ropi or --apcs /rwpi, for example:

```
int i;              // rw
int *p1 = &i;       // this static initialization does not work
                    // with --apcs /rwpi --no_lower_rwpi
extern const int ci; // ro
const int *p2 = &ci; // this static initialization does not work
                    // with --apcs /ropi
```

However, to enable these static initializations to work, use the --lower_rwpi and --lower_ropi options (see *Defining optimization criteria* on page 2-43).

To compile this code, type:

```
armcc --apcs /rwpi/ropi --lower_ropi
```

You do not have to specify --lower_rwpi, because this is the default.

- The use of --apcs /fpic is supported when compiling C++. Here, virtual table functions and typeinfo are placed in read-write areas so that they can be accessed relative to the location of the PC.

- If you use --apcs /fpic , the compiler exports only functions and data marked dllexport.

- If you use --apcs /fpic and --no_hide_all on the same command line, the compiler ignores __declspec(dll*) and exports all extern variables and functions. The compiler disables auto-inlining for exported functions.

- If you use --apcs /fpic in GNU mode, you must also use --no_hide_all.

Stack checking qualifiers

These --apcs qualifiers control software stack checking:

<u>/noswst</u>ackcheck   Uses the non-software stack checking AAPCS variant. This is the default.

<u>/swst</u>ackcheck   Uses the software stack checking AAPCS variant.

          ARM DUI 0282B

Chapter 3
ARM Compiler Reference

This chapter gives reference information on the ARM® compiler. It contains the following sections:

- *Compiler-specific features* on page 3-2
- *Language extensions* on page 3-30
- *C and C++ implementation details* on page 3-47
- *GNU extensions to the ARM compiler* on page 3-66
- *Predefined macros* on page 3-89.

For additional reference material on the ARM compiler see also:

- Appendix B *Standard C Implementation Definition*
- Appendix C *Standard C++ Implementation Definition*
- Appendix D *C and C++ Compiler Implementation Limits*.

3.1 Compiler-specific features

This section describes the ARM compiler-specific features, and includes:

- *Pragmas*
- *Function keywords* on page 3-9
- *Operators* on page 3-14
- *Intrinsic functions* on page 3-15
- *Variable declaration keywords* on page 3-21
- *Eight-byte alignment features* on page 3-28.

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

——— **Note** ———

Features described here are outside the ISO specification and might not easily port to other compilers.

3.1.1 Pragmas

Pragmas of the following form are recognized by the ARM compiler:

`#pragma [no_]feature-name`

——— **Note** ———

Pragmas override the related command-line options. For example, `#pragma arm` overrides the `--thumb` command-line option.

Pragmas are listed in Table 3-1. The following sections describe these pragmas in more detail.

Table 3-1 Pragmas recognized by the ARM compiler

Pragma name	Default	Reference
`anon_unions`	Off	*Pragmas controlling anonymous structures and unions* on page 3-8
`arm`	–	*Pragmas controlling code generation* on page 3-5
`arm section`	Off	*Pragmas controlling code generation* on page 3-5
`thumb`	–	*Pragmas controlling code generation* on page 3-5

Table 3-1 Pragmas recognized by the ARM compiler

Pragma name	Default	Reference
[no_]check_printf_formats	Off	*Pragmas controlling printf and scanf argument checking* on page 3-4
[no_]check_scanf_formats	Off	*Pragmas controlling printf and scanf argument checking* on page 3-4
[no_]check_stack	Off	*Pragmas controlling code generation* on page 3-5
[no_]debug	Off	*Pragmas controlling debugging* on page 3-5
diag_default	–	*Pragmas controlling diagnostic messages* on page 3-8
diag_error	–	*Pragmas controlling diagnostic messages* on page 3-8
diag_remark	–	*Pragmas controlling diagnostic messages* on page 3-8
diag_suppress	–	*Pragmas controlling diagnostic messages* on page 3-8
diag_warning	–	*Pragmas controlling diagnostic messages* on page 3-8
[no_]exceptions_unwind	–	*Pragmas controlling code generation* on page 3-5
hdrstop	–	*Pragmas controlling PCH processing* on page 3-8
import	–	*Pragmas controlling code generation* on page 3-5
no_pch	–	*Pragmas controlling PCH processing* on page 3-8
once	–	*Pragmas controlling code generation* on page 3-5
O*num*	–	*Pragmas controlling multiple optimizations* on page 3-5
Ospace	–	*Pragmas controlling multiple optimizations* on page 3-5
Otime	–	*Pragmas controlling multiple optimizations* on page 3-5

Table 3-1 Pragmas recognized by the ARM compiler

Pragma name	Default	Reference
pop	–	*Pragmas for saving and restoring the pragma state*
push	–	*Pragmas for saving and restoring the pragma state*
[no_]softfp_linkage	Off	*Pragmas controlling code generation* on page 3-5

Pragmas for saving and restoring the pragma state

The following pragmas enable you to save and restore the pragma state:

push   Saves the current pragma state.

pop   Restores the previously saved pragma state.

Pragmas controlling printf and scanf argument checking

The following pragmas control type checking of printf-like and scanf-like arguments:

check_printf_formats no_check_printf_formats

> Marks **printf**-like functions for type checking against a literal format
> string, if it exists. If the format is not a literal string, no type checking is
> done. The format string must be the last fixed argument. For example:

```
#pragma check_printf_formats
extern void myprintf(const char * format,...);
                    //printf format
#pragma no_check_printf_formats
```

check_scanf_formats no_check_scanf_formats

> Marks a function declared as a **scanf**-like function, so that the arguments
> are type checked against the literal format string. If the format is not a
> literal string, no type checking is done. The format string must be the last
> fixed argument. For example:

```
#pragma check_scanf_formats
extern void myformat(const char * format,...);
                    //scanf format
#pragma no_check_scanf_formats
```

Pragmas controlling debugging

The following pragma controls aspects of debug table generation:

debug no_debug Turns debug table generation on or off.

> If #pragma no_debug is specified, no debug table entries are generated for subsequent declarations and functions until the next #pragma debug.

Pragmas controlling multiple optimizations

These pragmas enable you to assign multiple optimizations on individual functions. The pragmas must be placed outside of a function, and you cannot apply more than one of these optimizations on a function. The following pragmas control these optimizations (see *Multi-optimization options* on page 2-43 for more information):

O*num*   Changes optimization level. The value of *num* is 0, 1, 2 or 3.

Ospace   Optimizes for space.

Otime   Optimizes for time.

Pragmas controlling code generation

The following pragmas control how code is generated (other code generation options are available from the compiler command line, see *Controlling code generation* on page 2-42):

arm   Switches code generation to the ARM instruction set. This pragma overrides the --thumb compiler option.

thumb   Switches code generation to the Thumb® instruction set. This pragma overrides the --arm compiler option.

> ——— **Note** ———
>
> If a module contains functions marked with #pragma arm or #pragma thumb, the module must be compiled with --apcs /interwork. This ensures that the functions can be called successfully from the other (ARM or Thumb) state.

check_stack no_check_stack

> #pragma check_stack re-enables the generation of function entry code that checks for stack limit violation if stack checking has been disabled with #pragma no_check_stack and the --apcs /swst command-line option is used (see *Stack checking qualifiers* on page 2-68).

exceptions_unwind no_exceptions_unwind

> Enables or disables function unwinding at runtime. See *Function unwinding at runtime* on page 3-63 for more details.

once
> When this is placed at the beginning of a header file, it indicates that the header file has been written in a way that including it several times has the same effect as including it once. Therefore, the compiler skips any subsequent includes of that file.
>
> Typically, you place a #ifndef guard around the body of the file, with a #define of the guard variable after the #ifndef, for example:
>
> ```
> #pragma once        // optional
> #ifndef FILE_H
> #define FILE_H... body of the header file ...#endif
> ```
>
> The #pragma once is marked as optional in this example. This is because the compiler recognizes the #ifndef header guard coding and skips subsequent includes even if #pragma once is absent.
>
> #pragma once is accepted for compatibility with other compilers, and enables you to use other forms of header guard coding. However, it is preferable to use #ifndef and #define coding because this is more portable.

softfp_linkage no_softfp_linkage

> #pragma softfp_linkage asserts that all function declarations up to the next #pragma no_softfp_linkage describe functions that use software floating-point linkage. The __softfp keyword has the same effect (see *Function keywords* on page 3-9). The pragma form can be useful when applied to an entire interface specification (header file) without altering that file.

import(*symbol_name*)

> Generates an importing reference to *symbol_name*. This is the same as the assembler directive:
>
> ```
>     IMPORT symbol_name
> ```
>
> The symbol name is placed in the symbol table of the image as an external symbol. It is otherwise unused. You must not define the symbol or make a reference to it.

You can use this pragma to select certain features of the C library, such as the heap implementation or real-time division. If a feature described in this book requires a symbol reference to be imported, the required symbol is specified. For an example, see *Avoiding the semihosting SWI* on page 5-24.

arm section *section_sort_list*

Specifies that the code or data section name is used for subsequent functions or objects. This includes definitions of anonymous objects the compiler creates for initializations. The option has no effect on:

- inline functions (and their local static variables)
- template instantiations (and their local static variables)
- elimination of unused variables and functions
- the order that definitions are written to the object file.

The full syntax for the pragma is:

```
#pragma arm section [sort_type[[=]"name"]] [,sort_type="name"]*
```

Where *name* is the name to use for the section, and *sort_type* is one of:

- code
- rodata
- rwdata
- zidata.

If *sort_type* is specified but *name* is not, the section name for *sort_type* is reset to the default value. Enter #pragma arm section on its own to restore the names of all object sections to their defaults. See Example 3-1.

Example 3-1 Section naming

```
int x1 = 5;                      // in .data (default)
int y1[100];                     // in .bss (default)
int const z1[3] = {1,2,3};       // in .constdata (default)
#pragma arm section rwdata = "foo", rodata = "bar"
int x2 = 5;                      // in foo (data part of region)
int y2[100];                     // in .bss
int const z2[3] = {1,2,3};       // in bar
char *s2 = "abc";                // s2 in foo, "abc" in .conststring
#pragma arm section rodata
int x3 = 5;                      // in foo
int y3[100];                     // in .bss
int const z3[3] = {1,2,3};       // in .constdata
char *s3 = "abc";                // s3 in foo, "abc" in .conststring
```

```
#pragma arm section code = "foo"
int add1(int x)                    // in foo (code part of region)
{
    return x+1;
}
#pragma arm section code
```

Use a scatter-loading description file with the ARM linker to control how to place a named section at a particular address in memory (see the chapter *Using Scatter-loading description file* in *RealView Developer Kit v2.2 Linker and Utilities Guide*).

Pragmas controlling PCH processing

The following pragmas control PCH processing:

hdrstop   Enables you to specify where the set of header files that are subject to precompilation ends.

             This must appear before the first token that does not belong to a preprocessing directive.

no_pch   Suppresses PCH processing for a given source file.

See *Precompiled header files* on page 2-14 for a detailed description of PCH files.

Pragmas controlling anonymous structures and unions

The following pragma controls the use of anonymous structures and unions:

anon_unions Enables support for anonymous structures and unions. See *Anonymous classes, structures and unions* on page 3-36 for more details.

Pragmas controlling diagnostic messages

The following pragmas control the output of the diagnostic messages that have a -D postfix in the message number:

diag_default *tag*[, *tag*, ...]

             Returns the severity of a diagnostic message to the one that was in effect before any pragmas were issued (that is, the normal severity of the message as modified by any command-line options).

diag_error *tag*[, *tag*, ...]

             Sets the diagnostic messages that have the specified tag(s) to Error severity.

```
diag_remark tag[, tag, ...]
```

> Sets the diagnostic messages that have the specified tag(s) to Remark severity.

```
diag_suppress tag[, tag, ...]
```

> Disables all diagnostic messages that have the specified tag(s).

```
diag_warning tag[, tag, ...]
```

> Sets the diagnostic messages that have the specified tag(s) to Warning severity.

For example, a Warning message might be displayed:

```
Warning:  #550-D: variable "b" was set but never used
```

If you want to lower this warning to a remark, specify:

```
#pragma diag_remark 550
```

By default, the compiler does not display remarks. To see the remark messages, use the `--remarks` compiler option (see *Controlling the output of diagnostic messages* on page 2-60).

3.1.2   Function keywords

Several keywords tell the compiler to give a function special treatment. These are all ARM extensions to the ISO C and C++ specifications, and are grouped in the following sections:

- *Function qualifiers*
- *Function storage class modifiers* on page 3-13.

Function qualifiers

Function qualifiers affect the type of a function. The qualifiers are placed after the parameter list in the same position that const and **volatile** can appear for C++ member function types.

```
__declspec(dllimport)
```

> Imports a symbol through the dynamic symbol table when building DLL libraries.

If this is applied to an inline function, the function definition (in this compilation unit) might be inlined, but will never be generated out-of-line. An out-of-line call or address reference will refer to the imported symbol.

`__declspec(dllexport)`

Exports the definition of a symbol through the dynamic symbol table when building DLL libraries.

If this is applied to an inline function, the function definition might be inlined, but an out-of-line instance of the function will always be generated, and exported, in the same way as for a non-inline function.

`__declspec(noreturn)`

Asserts that a function never returns. This reduces the cost of calling functions that never return, such as `exit()`. If a noreturn function returns to its caller, the behavior is undefined. The following example shows how a function that calls the noreturn function `overflow()` generates more efficient code:

```
__declspec(noreturn) void overflow(void);
int negate(int x)
{
    if (x == 0x80000000) overflow();
    return -x;
}
```

When using the `__declspec(noreturn)` qualifier, you must be aware that the improved code size might result from not preserving the return address when calling the function. This optimization limits the ability of a debugger to display the call-stack.

`__irq`   Enables a C or C++ function to be used as an interrupt routine called by the IRQ or FIQ vectors. All corrupted registers except floating-point registers are preserved, not only those that are normally preserved under the AAPCS. The default AAPCS mode must be used.

The function exits by setting the PC to `lr–4` and the CPSR to the value in SPSR. No arguments or return values can be used with `__irq` functions.

——— **Note** ———

When compiling for Thumb (`--thumb` option or `#pragma thumb`), any functions specified as `__irq` are compiled for ARM.

__pure   Asserts that a function declaration is pure. Functions that are pure are candidates for common subexpression elimination. By default, functions are assumed to be impure (causing side-effects). A function is properly defined as pure only if:

- its result depends exclusively on the values of its arguments
- it has no side effects, for example it cannot call impure functions.

So, a pure function cannot use global variables or dereference pointers, because the compiler assumes that the function does not access memory (except stack memory) at all. When called twice with the same parameters, a pure function must return the same value each time.

The __pure declaration can also be used as a prefix or postfix declaration. In some cases the prefix form can be ambiguous and readability is improved by using the postfix form:

```
__pure void (*h(void))(void);  /* declares 'h' as a (pure?)
function that returns a pointer to a (pure?) function.  It is
ambiguous which of the two function types is pure. */
void (*h1(void) __pure)(void);  /* 'h1' is a pure function
returning a pointer to a (normal) function */
```

__softfp   Asserts that a function uses software floating-point linkage. Calls to the function pass floating-point arguments in integer registers. If the result is a floating-point value, the value is returned in integer registers. This duplicates the behavior of compilation targeting software floating-point.

This keyword enables an identical library to be used by sources compiled to use hardware and software floating-point.

__swi   Declares a SWI function taking up to four integer-like arguments and returning up to four results in a value_in_regs structure. This causes function invocations to be compiled inline as an AAPCS compliant SWI that behaves similarly to a normal call to a function.

For a SWI returning no results use:

```
void __swi(swi_num) swi_name(int arg1,…, int argn);
```
For example:

```
void __swi(42) terminate_proc(int procnum);
```
For a SWI returning one result, use:

```
int __swi(swi_num) swi_name(int arg1,…, int argn);
```
For a SWI returning more than 1 result use:

```
typedef struct res_type { int res1,…,resn;} res_type;
res_type __value_in_regs __swi(swi_num) swi_name(
            int arg1,…,int argn);
```

The `__value_in_regs` qualifier is used to specify that a small structure of up to four words (16 bytes) is returned in registers, rather than by the usual structure-passing mechanism defined in the AAPCS.

__swi_indirect

Passes an operation code to the SWI handler in r12:

```
int __swi_indirect(swi_num)
            swi_name(int real_num, int arg1, … argn);
```

where:

real_num   Is the SWI number used in the SWI instruction.

real_num   Is the value passed in r12 to the SWI handler. You can use this feature to implement indirect SWIs. The SWI handler can use r12 to determine the function to perform.

For example:

```
int __swi_indirect(0) ioctl(int swino, int fn, void *argp);
```

This SWI can be called as follows:

```
ioctl(IOCTL+4, RESET, NULL);
```

It compiles to a SWI 0 with `IOCTL+4` in r12.

To use the indirect SWI mechanism, your system SWI handlers must make use of the r12 value to select the required operation.

__swi_indirect_r7

`__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, for example:

```
long __swi_indirect_r7(0) SWI_write(unsigned, int fd, const char *
buf, size_t count);
#define write(fd, buf, count) write(4, (fd), (buf), (count))
```

Calling

```
    write(fd, buf, count);
```

compiles to a SWI 0 with r0 = fd, r1 = buf, r2 = count, and r7 = 4.

__value_in_regs

Instructs the compiler to return a structure of up to four integer words in integer registers or up to four floats or doubles in floating-point registers rather than using memory, for example:

```
typedef struct int64_struct {
    unsigned int lo;
    unsigned int hi;
} int64_struct;
__value_in_regs extern
    int64_struct mul64(unsigned a, unsigned b);
```

Declaring a function __value_in_regs can be useful when calling assembler functions that return more than one result.

——— **Note** ———

A C++ function cannot return a __value_in_regs structure if the structure requires copy constructing.

Where the structure returned in a __value_in_regs function is too big, a warning is produced and that __value_in_regs is then ignored.

Function storage class modifiers

A storage class modifier is a subset of function declaration keywords, however they do not affect the type of the function.

__asm *return-type function-name*(*param-list*) {*assembler-code*}

Instructs the compiler that the function contains only assembler language. This function is embedded in your C or C++ source, see *Embedded assembler syntax* on page 4-20 for more details on embedded assembler. The alternative **asm** keyword is only accepted in C++.

__forceinline

Forces the compiler to compile a C or C++ function inline. The compiler attempts to inline the function, regardless of the characteristics of the function. However, the compiler does not inline a function if doing so causes problems, for example, a recursive function is inlined only once. The semantics of __forceinline are exactly the same as those of the C++ **inline** keyword. See *Defining optimization criteria* on page 2-43 for information on command-line options that affect inlining.

__inline   Suggests that the compiler compiles a C or C++ function inline, if it is sensible to do so. The semantics of __inline are exactly the same as those of the C++ **inline** keyword:

```
__inline int f(int x) {return x*5+1;}
int g(int x, int y) {return f(x) + f(y);}
```

The compiler compiles functions inline when __inline is used and the functions are not too large. Large functions are not compiled inline because they can adversely affect code density and performance. See *Defining optimization criteria* on page 2-43 for information on command-line options that affect inlining.

3.1.3 Operators

The following functions are ARM language extensions to the ISO C and C++ specifications:

- *__ALIGNOF__*
- *__INTADDR__*.

These extensions are always available.

__ALIGNOF__

__ALIGNOF__ is similar to sizeof, but returns either an alignment requirement value for a type, or 1 if there is no alignment requirement. It can be followed by a *type* or *expression* in parentheses:

```
__ALIGNOF__(type)
__ALIGNOF__(expression)
```

——— **Note** ———

The expression in the second form is not evaluated.

__INTADDR__

__INTADDR__(*expression*) treats the enclosed expression as a constant expression, and converts it to an integer constant.

——— **Note** ———

This is used in the offsetof macro.

3.1.4   Intrinsic functions

The following intrinsic functions are ARM language extensions to the ISO C and C++ specifications:

- *__breakpoint() intrinsic*
- *__current_pc() intrinsic* on page 3-16
- *__current_sp() intrinsic* on page 3-16
- *__return_address() intrinsic* on page 3-16
- *__nop() intrinsic* on page 3-17
- *Intrinsics that modify the CPSR* on page 3-18.

These extensions are always available.

__breakpoint() intrinsic

The __breakpoint() intrinsic enables you to include an ARM or Thumb BKPT instruction in your C or C++ code. You cannot use this intrinsic in inline assembly. The syntax for this intrinsic is:

```
__breakpoint(value);
```

value is a compile-time constant integer, and must be included. The value must be an integer in the range:

0 to 65535   if compiled as ARM code.

0 to 255   if compiled as Thumb code.

For example:

```
func() {
    ...
    __breakpoint(0xF02C);
    ...
}
```

If __breakpoint() is used on architectures that do not support the BKPT instruction, a warning is generated. If a BKPT instruction is executed on an architecture that does not support it, the undefined instruction trap is taken.

The exact location of the BKPT instruction is not guaranteed unless you specify the minimum optimization level -O0. At all other optimization levels the compiler might move, but not remove, the BKPT instruction.

For more details of the BKPT instruction, see *RealView Developer Kit v2.2 Assembler Guide*.

__current_pc() intrinsic

Returns the value of the program counter (PC) at the point where __current_pc() is used. The implicit prototype for this intrinsic is:

```
int __current_pc(void)
```

—— **Note** ——

If you have legacy source that accesses the pc register from inline assembly, see *Legacy inline assembler that accesses sp, lr or pc* on page 4-29.

__current_sp() intrinsic

Returns the value of the stack pointer (sp) at the point where __current_sp() is used. The implicit prototype for this intrinsic is:

```
int __current_sp(void)
```

—— **Note** ——

If you have legacy source that accesses the sp register from inline assembly, see *Legacy inline assembler that accesses sp, lr or pc* on page 4-29.

__return_address() intrinsic

Returns the value of the link register (lr) that is used to return from the current function. The implicit prototype for this intrinsic is:

```
int __return_address(void)
```

—— **Note** ——

If you have legacy source that accesses the lr register from inline assembly, see *Legacy inline assembler that accesses sp, lr or pc* on page 4-29.

The `__return_address()` intrinsic does not affect the ability of the compiler to perform optimizations, such as inlining, tail calling, and code sharing. However, the returned value reflects the optimizations performed. For more details on the optimization options, see *Defining optimization criteria* on page 2-43.

Different optimizations are applied as follows:

Inlining enabled

> If a function call is inlined then the value returned by `__return_address()` is the return address for the function that calls the inlined function.

Inlining and tail calling disabled

> If inlining and tail calling are disabled, the value returned by `__return_address()` is the return address of the next instruction in the calling function.

Inlining disabled and tail calling enabled

> If a function call is inlined and tail calling is enabled, then the value returned by `__return_address()` within the called function is the return address for the caller function.

__nop() intrinsic

The `__nop()` intrinsic is used to insert a single NOP instruction into the instruction stream generated by the compiler. This enables you to, for example, insert a small delay between reading and writing to a memory-mapped peripheral. The implicit prototype for this intrinsic is:

```
void __nop(void)
```

One NOP instruction is generated for each `__nop()` in the source. The compiler does not optimize-away the NOP instructions, except for normal unreachable-code elimination. `__nop()` also acts as a barrier for instruction scheduling in the compiler, that is instructions are not moved from one side of the NOP to the other as a result of optimization.

──── **Note** ────

You can use the `__schedule_barrier()` intrinsic to insert a scheduling barrier without generating a NOP instruction. See *Optimization barrier intrinsics* on page 3-19 for more information.

────────────

For example:

```
volatile int *hw_reg1;
volatile int *hw_reg2;
int update(int input)
{
    /* update the peripheral */
    *hw_reg1 = input;
    /* allow peripheral time to update its status */
    __nop();
    /* read new status */
    return *hw_reg2;
}
```

Intrinsics that modify the CPSR

These functions enable you to control IRQ and FIQ interrupts (see *Enabling and disabling interrupts* on page 4-6).

The prototypes for these intrinsics are:

```
void __disable_irq(void);
void __enable_irq(void);
void __disable_fiq(void);
void __enable_fiq(void);
```

`__{dis,en}able_{ir,fi}q` can only be executed in privileged modes, that is, in non-user modes. In User mode these instructions do not change the interrupt flags in the CPSR.

Architectures

These intrinsics are available for all processor architectures in both ARM and Thumb state:

- If you are compiling for processors that support ARMv6 (or later), a CPS instruction is generated inline for these functions, for example:

    ```
    CPSID  i
    ```

- If you are compiling for processors that support ARMv4 or ARMv5 in ARM state, the compiler inlines a sequence of MRS and MSR instructions, for example:

    ```
    MRS  r0, CPSR
    ORR  r0, r0, #0x80
    MSR  CPSR_c, r0
    ```

- If you are compiling for processors that support ARMv4 or ARMv5 in Thumb state, the compiler calls a helper function, for example:

```
BL      __ARM_disable_irq
```

For more information on these instructions, see *RealView Developer Kit v2.2 Assembler Guide*.

Restrictions

You cannot use these functions to change any other CPSR bits including the mode, state, and imprecise data abort setting. This means that the intrinsics can be used only if the processor is already in a privileged mode, because the control bits of the CPSR and SPSR cannot be changed in User mode.

Optimization barrier intrinsics

The ARM compiler can perform a range of optimizations, including re-ordering instructions and merging some operations. In some cases, such as system level programming where memory is being accessed concurrently by multiple processes, it might be necessary to disable instruction re-ordering and force memory to be updated.

The following optimization barrier intrinsics do not generate code, however they can result in slightly increased code size and additional memory accesses:

```
void __schedule_barrier(void);
void __force_stored(void);
void __memory_changed(void):
```

———— **Note** ————

On some systems the memory barrier intriniscs might not be sufficient to ensure memory consistency. For example, the `__memory_changed()` intrinsic forces values held in registers to be written out to memory. However, if the destination for the data is held in a bufferable region it might wait in a write buffer. In this case you might also have to write to CP15 to drain the write buffer. Refer to the Technical Reference Manual for your ARM core for more information.

————————————

`void __schedule_barrier(void)`

> A scheduling barrier creates a sequence point where operations before and operations after the sequence point are not merged by the compiler. A scheduling barrier does not cause memory to be updated. If variables are held in registers they are updated in place, and not written out. This intrinsic is similar to the `__nop()` intrinsic, except that no NOP instruction is generated.

void __force_stores(void)

> A force stores barrier causes all variables that are visible outside the current function, such as variables that have pointers to them passed into or out of the function, to be written back to memory if they have been changed. The `__force_stores()` intrinsic also acts as a scheduling barrier.

void __memory_changed(void)

> A memory changed barrier causes all variables that are visible outside the current function, such as variables that have pointers to them passed into or out of the function, to be written back to memory if they have been changed, and then to be read back from memory. The `__memory_changed()` intrinsic also acts as a scheduling barrier.

3.1.5   Variable declaration keywords

This section describes the implementation of various standard variable declaration keywords, and those that are specific to the ARM compiler. ISO C or C++ Standard keywords that do not have behavior or restrictions that are specific to the ARM compiler are not documented. See also *Type qualifiers* on page 3-26 for information on qualifiers such as **volatile** and **__packed**.

The keywords are grouped in the following sections:
*   *Standard keywords*
*   *Keywords specific to the ARM compiler* on page 3-22
*   *Type qualifiers* on page 3-26.

Standard keywords

These keywords declare a storage class.

register   Declares a local object (auto variable) to have the storage class **register**. A **register** declaration is a suggestion to the compiler that it is to attempt to put the object in a physical register. However, if no more registers are available, the compiler places the object on the stack instead.

> —— **Note** ——
>
> It is recommended that you do not use **register**, because the compiler automatically makes the best choice. Also, if you do use **register**, it restricts the use of registers for optimization, and this might increase the code size.

Depending on the variant of the AAPCS being used, there are between five and seven integer registers available, and four floating-point registers. In general, it is recommended that you do not declare more than four integer register variables and two floating-point register variables.

The following object types can be declared to have the **register** storage class:

*   All integer types (**long long** occupies two registers).
*   All integer-like structures. That is, any one word **struct** or **union** where all addressable fields have the same address, or any one word structure containing bitfields only. The structure must be padded to 32 bits.
*   Any pointer type.

- Floating-point types. The double-precision floating-point type **double** occupies two ARM registers if software floating-point is used.

Keywords specific to the ARM compiler

The keywords in this section are used to declare or modify variable definitions:

__int64　　This type specifier is an alternative name for type **long long**. This is accepted even when using --strict.

__global_reg(*vreg*)

This storage class specifier allocates the declared variable to a global integer register variable. If you use this storage class, you cannot also use any of the other storage classes such as **extern**, **static**, or **typedef**. *vreg* is an AAPCS callee-save register (for example, v1) and not a real register number (for example, r4). In C, global register variables cannot be qualified or initialized at declaration. In C++, any initialization is treated as a dynamic initialization. Valid types are:

- any integer type, except **long long**
- any pointer type.

For example, to declare a global integer register variable allocated to r5 (the AAPCS register v2), use the following:

```
__global_reg(2) int x;
```

The global register must be specified in all declarations of the same variable. For example, the following is an error:

```
int x;
__global_reg(1) int x; // error
```

Also, __global_reg variables in C cannot be initialized at definition. For example, the following is an error in C, though not in C++:

```
__global_reg(1) int x=1; // error in C
```

Depending on the AAPCS variant used, between five and seven integer registers, and four floating-point registers are available for use as global register variables. In practice, it is recommended that you do not use more than three global integer register variables in ARM code, or more than one global integer register variable in Thumb code, or more than two global floating-point register variables.

———— **Note** ————

In Thumb, `__global_reg(4)` is not valid.

————————————

Unlike register variables declared with the standard **register** keyword, the compiler does *not* move global register variables to memory as required. If you declare too many global variables, code size increases significantly. In some cases, your program might not compile.

———— **Caution** ————

You must take care when using global register variables because:

- There is no check at link time to ensure that direct calls between different compilation units are sensible. If possible, define global register variables used in a program in each compilation unit of the program. In general, it is best to place the definition in a global header file. You must set up the value in the global register early in your code, before the register is used.

- A global register variable maps to a callee-saved register, so its value is saved and restored across a call to a function in a compilation unit that does not use it as a global register variable, such as a library function.

- Calls back into a compilation unit that uses a global register variable are dangerous. For example, if a global register using function is called from a compilation unit that does not declare the global register variable, the function reads the wrong values from its supposed global register variables.

- This class can only be used at file scope.

————————————

`__align(n)`   The `__align(n)` storage class modifier aligns a top-level object on an *n*-byte boundary, where *n* is either 2, 4, or 8. Eight-byte alignment is required if you are using the LDRD or STRD instructions, and can give a significant performance advantage with VFP instructions.

For example, if you are using LDRD or STRD instructions to access data objects defined in C or C++ from ARM assembly language, you must use the `__align(8)` storage class specifier to ensure that the data objects are properly aligned.

You can specify a power of two for the alignment boundary, however eight is the maximum for auto variables. Also, you can only overalign. That is you can make a two-byte object four-byte aligned, but you cannot align a four-byte object at two bytes.

__align(*n*) is a storage class modifier. This means that it can be used only on top-level objects. You cannot use it on:

- types, including typedefs and structure definitions
- function parameters.

It can be used in conjunction with **extern** and **static**.

__align(8) only ensures that the qualified object is eight-byte aligned. This means, for example, that you must explicitly pad structures if required.

__weak

The __weak keyword can be applied to function and object declarations, and to function definitions:

Function and object declarations

For declarations, this storage class specifies an **extern** object declaration that, if not present, does not cause the linker to fault an unresolved reference.

If the reference is made from code that compiles to a branch or branch link instruction, the reference is resolved as branching to the next instruction. This makes the branch a NOP:

```
__weak void f(void);
...
f(); // call f weakly
```

A function or object cannot be used both weakly and nonweakly in the same compilation. For example the following code uses f() weakly from g() and h():

```
void f(void);
void g() {f();}
__weak void f(void);
void h() {f();}
```

It is not possible to use a function or object weakly from the same compilation that defines the function or object. The following code uses f() nonweakly from h():

```
__weak void f(void);
void h() {f();}
void f() {}
```

The linker does not load the function or object from a library unless another compilation uses the function or object nonweakly. If the reference remains unresolved, its value is assumed to be NULL. Unresolved references, however, are not

---

NULL if the reference is from code to a position-independent section or to a missing __weak function (see Example 3-2 on page 3-25).

Example 3-2 Non-NULL unresolved references

```
__weak const int c;              // assume 'c' is not present in final link
const int *f1() { return &c; }   // '&c' will be non-NULL if
                                 // compiled and linked /ropi
__weak int i;                    // assume 'i' is not present in final link
int *f2() { return &i; }         // '&i' will be non-NULL if
                                 // compiled and linked /rwpi
__weak void f(void);             // assume 'f' is not present in final link
typedef void (*FP)(void);
FP g() { return f; }             // 'g' will return non-NULL if
                                 // compiled and linked /ropi
```

See the chapter on creating and using libraries in *RealView Developer Kit v2.2 Linker and Utilities Guide* for details on library searching.

Function definitions

Functions defined with __weak export their symbols weakly. A weakly defined function behaves like a normally defined function unless a non-weakly defined function of the same name is linked into the same image.If both a non-weakly defined function and a weakly defined function exist in the same image then all calls to the function resolve to call the non-weak function. If multiple weak definitions are available, the linker chooses one for use by all calls.

Weakly defined functions cannot be inlined.

Functions declared with __weak and then defined without __weak behave as non-weak functions.

Type qualifiers

This section describes the implementation of various standard C type qualifiers and type qualifiers specific to the ARM compiler. These type qualifiers can be used to instruct the compiler to treat the qualified type in a special way. Standard qualifiers that do not have behavior or restrictions that are specific to the ARM compiler are not documented:

__packed   The __packed qualifier sets the alignment of any valid type to 1. This means:

• there is no padding inserted to align the packed object

• objects of packed type are read or written using unaligned accesses.

The __packed qualifier cannot be used on structures that were previously declared without __packed.

——— **Note** ———

__packed is not, strictly, a type qualifier. It is included in this section because it behaves like a type qualifier in most respects.

The __packed qualifier does not affect local variables of integral type.

The __packed qualifier applies to all members of a structure or union when it is declared using __packed. There is no padding between members, or at the end of the structure. All substructures of a packed structure must be declared using __packed. Integral subfields of an unpacked structure can be packed individually.

A packed structure or union is not assignment-compatible with the corresponding unpacked structure. Because the structures have a different memory layout, the only way to assign a packed structure to an unpacked structure is by a field-by-field copy. See also *Packed structures* on page 3-55.

The effect of casting away __packed is undefined. The effect of casting a non-packed structure to a packed structure is undefined. A pointer to an integral type can be legally cast, explicitly or implicitly, to a pointer to a packed integral type.

A pointer can point to a packed type (see Example 3-3).

Example 3-3 Pointer to packed

```
typedef __packed int* PpI;         /* pointer to a __packed int */
PpI p2;                            /* 'p2' has the same type as 'p' */
                                   /* __packed is a qualifier */
                                   /* just like 'const' or 'volatile' */
```

```
typedef int *PI;                    /* pointer to int */
__packed PI p3;                     /* a __packed pointer to a normal int */
                                    /* -- not the same type as 'p' and 'p2' */
int *__packed p4;                   /* 'p4' has the same type as 'p3' */
```

There are no packed array types. A packed array is an array of objects of packed type. There is no padding in the array.

——— **Note** ———

On ARM processors, access to unaligned data can take up to seven instructions and three work registers. Data accesses through packed structures must be minimized to avoid increase in code size and performance loss.

The `__packed` qualifier is useful to map a structure to an external data structure, or for accessing unaligned data, but it is generally not useful to save data size because of the relatively high cost of access. The number of unaligned accesses can be reduced by only packing fields in a structure that requires packing.

When a packed object is accessed using a pointer, the compiler generates code that works and that is independent of the pointer alignment (see Example 3-4).

Example 3-4 Packed structure

```
typedef __packed struct
{
    char x;        // all fields inherit the __packed qualifier
    int y;
}X;        // 5 byte structure, natural alignment = 1
int f(X *p)
{
    return p->y;    // does an unaligned read
}
typedef struct
{
    short x;
    char y;
    __packed int z; // only pack this field
    char a;
}Y;    // 8 byte structure, natural alignment = 2
int g(Y *p)
```

```
{
    return p->z + p->x;     // only unaligned read for z
}
```

---

volatile   The standard ISO qualifier **volatile** informs the compiler that the
              qualified type contains data that can be changed from outside the
              program. Such variables must be placed in RW or ZI data (not in
              .constdata) so that they can be changed. This includes **const volatile**
              variables.

              The compiler does not attempt to optimize accesses to **volatile** types. For
              example, volatile structures can be mapped onto memory-mapped
              peripheral registers:

```
/* define a memory-mapped port register */
volatile unsigned *port = (unsigned int *) 0x40000000;
/* to access the port */
*port = value;     /* write to port */
value = *port;     /* read from port */
```

              In ARM C and C++, a **volatile** object is accessed if any word or byte (or
              halfword on ARM architectures with halfword support) of the object is
              read or written. For **volatile** objects, reads and writes occur as directly
              implied by the source code, in the order implied by the source code. The
              effect of accessing a **volatile short** is undefined for ARM architectures
              that do not support halfwords. Accessing **volatile** packed data is
              undefined.

3.1.6   Eight-byte alignment features

The following list summarizes the eight-byte alignment features of the ARM compiler:

• In *RealView® Compilation Tools* (RVCT) v2.x, all code is compiled with the
  REQUIRE8 and PRESERVE8 directives. Therefore, check that your existing assembly
  files, object files, or libraries preserve eight-byte alignment, and correct them if
  required. For more details, see *RealView Developer Kit v2.2 Assembler Guide* and
  RealView Developer Kit v2.2 Linker and Utilities Guide.

• In RVCT v2.x, **double** and **long long** data types are eight-byte aligned. This
  enables efficient use of the LDRD and STRD instructions in ARMv5TE and later.

  In RVCT v2.2, the compiler supports the option --apcs /adsabi to compile code
  that is compatible with the old ADS ABI (see *ADS ABI qualifier* on page 2-64).
  However, this is deprecated and will be removed in a future release.

- If you are using LDRD or STRD instructions to access data objects defined in C or C++ from ARM assembly language, you must use the __align(8) storage class specifier to ensure that the data objects are properly aligned.

  __align(8) only ensures that the qualified object is eight-byte aligned. Therefore, you must explicitly pad structures if required.

  See *Keywords specific to the ARM compiler* on page 3-22 for more details on the __align(*n*) specifier.

- The *Procedure Call Standard for the ARM Architecture* (AAPCS) requires that the stack is eight-byte aligned at all external interfaces. The ARM compiler and C libraries ensure that eight-byte alignment of the stack is maintained. In addition, the default C library memory model maintains eight-byte alignment of the heap.

- The default implementations of malloc(), realloc(), and calloc() maintain an eight-byte aligned heap.

  See *Tailoring storage management* on page 5-77 for details on the heap implementations provided by the ARM compiler, and how to write your own heap implementation.

- The default implementation of alloca() returns an eight-byte aligned block of memory. See *alloca( )* on page 5-124 for more details on this C library extension.

3.2 Language extensions

This section describes the language extensions supported by the ARM compiler, and includes:

- *C and C++ language extensions*
- *C language extensions* on page 3-38
- *C++ language extensions* on page 3-44.

3.2.1 C and C++ language extensions

The compiler supports the following extensions to both the ISO C and the ISO/IEC C++ Standard:

- *Identifiers* on page 3-31
- *Integral type extensions* on page 3-31
- *Floating-point extensions* on page 3-31
- *Structure, union, enum, and bitfield extensions* on page 3-31
- *Void returns and arguments* on page 3-33
- *long long* on page 3-33
- *friend* on page 3-34
- *restrict* on page 3-34
- *Hexadecimal floating-point constants* on page 3-35
- *Pointer assignment and ROPI* on page 3-35
- *Scalar type constants* on page 3-35
- *Declaration of a class member* on page 3-35
- *Anonymous classes, structures and unions* on page 3-36
- *Type conversions* on page 3-37
- *? operator* on page 3-37
- *Arguments to functions* on page 3-37
- *Non-static local variables* on page 3-37
- *Keywords* on page 3-37
- *Other supported extensions* on page 3-37.

None of these extensions is available if the compiler is restricted to compiling strict 1990 ISO Standard C or strict ISO/IEC C++, for example, if you specify the --strict compiler option.

Identifiers

Dollar ($) signs are permitted in identifiers by default, unless you specify the `--strict` option. To permit dollar signs in identifiers with the `--strict` option, then also use the `--dollar` command-line option.

Integral type extensions

The following integral type extensions are supported:

• In an integral constant expression, an integral constant can be cast to a pointer type and then back to an integral type.

• In duplicate size and sign specifiers, for example, `short short` or `unsigned unsigned`, the following error is issued:

```
Error:  #240: duplicate specifier in declaration
```

Floating-point extensions

The following floating-point extensions are supported:

• `long float` is accepted as a synonym for `double`.

Structure, union, enum, and bitfield extensions

The following structure, union, enum, and bitfield extensions are supported:

• The last member of a `struct` can have an incomplete array type. It must not be the only member of the `struct`, otherwise the `struct` would have zero size.

————— **Note** —————
This is also permitted in C++, but only when the structure is C-like.

• The element type of a file-scope array can be an incomplete `struct`, `union`, or `enum` type. The type must be completed before the array is subscripted (if it is), and by the end of the compilation if the array is not `extern`. In C++, an incomplete class is also permitted.

• The final semicolon preceding the closing brace } of a `struct` or `union` specifier can be omitted. The following warning is issued:

```
#65-D: expected a ";"
```

- An initializer expression that is a single value and is used to initialize an entire static array, **struct**, or **union** does not have to be enclosed in braces. ISO C requires the braces.

- An extension is supported to enable constructs similar to C++ anonymous unions, including the following:

  — not only anonymous unions but also anonymous structs are permitted, that is, their members are promoted to the scope of the containing struct and looked up like ordinary members

  — they can be introduced into the containing **struct** by a **typedef** name - they do not have to be declared directly, as with true anonymous unions

  — a tag can be declared (C mode only).

  To enable support for anonymous structures and unions, you must use the anon_unions pragma (see *Pragmas controlling anonymous structures and unions* on page 3-8).

- An extra comma is permitted at the end of an **enum** list. The following remark is issued:

  ```
  #228-D: trailing comma is nonstandard
  ```

- **enum** tags can be incomplete. You can define the tag name and resolve it later, by specifying the brace-enclosed list.

- The values of enumeration constants can be given by expressions that evaluate to unsigned quantities that fit in the **unsigned int** range but not in the **int** range. For example:

  ```
  /* When ints are 32 bits: */
  enum a {w = -2147483648};  /* No error */
  enum b {x = 0x80000000};   /* No error */
  enum c {y = 0x80000001};   /* No error */
  enum d {z = 2147483649};   /* Error */
  ```

  The following error is issued:

  ```
  #66: enumeration value is out of "int" range
  ```

  See *Structures, unions, enumerations, and bitfields* on page 3-53 for full details on how enumerations are handled when out of range.

- Bit fields can have base types that are **enum** types or integral types besides **int** and **unsigned int**.

Void returns and arguments

Any **void** type, including a typedef to **void**, is permitted as the return type in a function declaration, or the indicator that a function takes no argument. For example, the following is permitted:

```
typedef void VOID;
int fn(VOID);       // Error in --strict C and C++
VOID fn(int x);     // Error in --strict C
```

long long

The ARM compiler supports 64-bit integer types through the type specifier **long long** and **unsigned long long**. They behave analogously to **long** and **unsigned long** with respect to the usual arithmetic conversions. **long long** is a synonym for __int64.

Integer constants can have:

- an ll suffix to force the type of the constant to **long long**, if it fits, or to **unsigned long long** if it does not fit

- a ull (or llu) suffix to force the type of the constant to **unsigned long long**.

Format specifiers for printf() and scanf() can include ll to specify that the following conversion applies to a **long long** argument, as in %lld or %llu.

Also, a plain integer constant is of type **long long** or **unsigned long long** if its value is large enough. There is a warning message from the compiler indicating the change. For example, in strict 1990 ISO Standard C 2147483648 has type **unsigned long**. In ARM C and C++ it has the type **long long**. One consequence of this is the value of an expression such as:

```
2147483648 > -1
```

The value of this expression is 0 in strict C and C++, and 1 in ARM C and C++.

The following restrictions apply to **long long**:

- **long long** enumerators are not available.

- The controlling expression of a **switch** statement cannot have (**unsigned**) **long long** type. Consequently case labels must also have values that can be contained in a variable of type **unsigned long**.

The **long long** types are accommodated in the usual arithmetic conversions.

friend

A **friend** declaration for a **class** can omit the class keyword, for example:

```
class B;
class A {
    friend B;  // Should be "friend class B"
};
```

Access checks are not carried out on **friend** declarations by default. Use the --strict command-line option to force access checking.

restrict

The **restrict** keyword is a C99 feature that enables you to ensure that different object pointer types and function parameter arrays do not point to overlapping regions of memory. Therefore, the compiler can perform optimizations that would otherwise be prevented because of possible aliasing. The keyword is available in both C and C++.

For example:

```
void copy_array(int n, int *restrict a, int *restrict b)
{
    while (n-- > 0)
        *a++ = *b++;
}
```

In this example, pointer *a does not point to the same region of memory as pointer *b. They can point to different arrays, or to different regions within an array:

```
void test(void)
{
    extern int array[100];
    copy_array(50, array + 50, array);    // valid
    copy_array(50, array + 1, array);     // undefined behavior
}
```

To enable the **restrict** keyword, you must specify the --restrict option (see *Controlling implementation details* on page 2-54). This keyword is supported in a form extended for C++. The set of C++ extensions is described in J16/92-0057.

Hexadecimal floating-point constants

The ARM compiler implements an extension to the syntax of numeric constants in C to enable explicit specification of floating-point constants as IEEE bit patterns. The syntax is:

0f_*n*   Interpret an 8-digit hex number *n* as a **float** constant. There must be exactly eight digits.

0d_*nn*   Interpret a 16-digit hex number *nn* as a **double** constant. There must be exactly 16 digits.

Pointer assignment and ROPI

Your code might contain constant expressions that assign pointers, for example:

```
extern const char a[];const char *p = a;
```

If you compile this with --apcs /ropi, the compiler gives the following warning in all modes:

```
Warning:  #1357-D: static initialisation of variable "p" using address of a may
cause link failure -ropi
```

Scalar type constants

Constants of scalar type can be defined within classes. This is an old form. The modern form uses an initialized static data member:

```
class A {
    const int size = 10;
    int a[size];
};
```

Declaration of a class member

In the declaration of a class member, a qualified name can be used:

```
struct A {
    int A::f();  // Should be int f();
};
```

Anonymous classes, structures and unions

Anonymous classes, structures, and unions are supported as an extension. Anonymous structures and unions are supported in C and C++.

Anonymous unions are available by default in C++. However, you must specify the anon_unions pragma if you want to use:
- anonymous unions and structures in C
- anonymous classes and structures in C++.

See *Pragmas controlling anonymous structures and unions* on page 3-8 for details.

An anonymous union can be introduced into a containing class by a **typedef** name. Unlike a true anonymous union, it does not have to be declared directly. For example:

```
typedef union {
    int i, j;
} U;                 // U identifies a reusable anonymous union.
#pragma anon_unions
class A {
    U;               // Okay -- references to A::i and A::j are allowed.
};
```

The extension also enables anonymous classes and anonymous structures, as long as they have no C++ features (for example, no static data members or member functions and no non-public members) and have no nested types except anonymous classes, structures, or unions. For example:

```
#pragma anon_unions
struct A {
    struct {
        int i, j;
    };               // Okay -- references to A::i and A::j are allowed.
};
```

Type conversions

Type conversion between a pointer to an `extern` `"C"` function and a pointer to an `extern` `"C++"` function is permitted, for example:

```
extern "C" void f();     // f's type has extern "C" linkage
void (*pf)()             // pf points to an extern "C++" function
             = &f;       // error unless implicit conversion is allowed
```

? operator

A ? operator whose second and third operands are string literals or wide string literals can be implicitly converted to `char *` or `wchar_t *`. (Recall that in C++ string literals are `const`. There is a deprecated implicit conversion that enables conversion of a string literal to `char *`, dropping the `const`. That conversion, however, applies only to simple string literals. Permitting it for the result of a ? operation is an extension.)

```
char *p = x ? "abc" : "def";
```

Arguments to functions

Default arguments can be specified for function parameters other than those of a top-level function declaration (for example, they are accepted on `typedef` declarations and on pointer-to-function and pointer-to-member-function declarations).

Non-static local variables

Non-static local variables of an enclosing function can be referenced in a non-evaluated expression (for example, a `sizeof` expression) inside a local class. A warning is issued.

Keywords

The ARM compiler implements some keyword extensions for functions and variables. See:

- *Function keywords* on page 3-9
- *Variable declaration keywords* on page 3-21.

Other supported extensions

An input file cannot contain declarations. For C and C++, a remark is issued.

If you specify the `--strict_warnings` option, this remark is upgraded to Warning. If you specify the `--strict` option, this remark is upgraded to Error.

3.2.2   C language extensions

The compiler supports the following extensions to the ISO C Standard:

- *// comments*
- *Constant expressions*
- *Array and pointer extensions* on page 3-40
- *Preprocessor extensions* on page 3-41
- *Keywords* on page 3-41
- *Other supported extensions* on page 3-41
- *Compiler behavior that is undefined by the ISO C Standard* on page 3-42.

None of these extensions is available if the compiler is restricted to compiling strict 1990 ISO Standard C, for example, if you specify the --strict compiler option.

// comments

The character sequence // starts a one line comment. As in C++, the comment is terminated by the next newline character.

——— **Note** ———

Comment removal takes place after line continuation, so:

```
// this is a - \
single comment
```

The characters of a comment are examined only to find the comment terminator, therefore:

- // has no special significance inside a comment introduced by /*
- /* has no special significance inside a comment introduced by //

Comment text can appear at the end of preprocessing directives.

Constant expressions

Extended constant expressions are supported in initializers. The following examples show the compiler behavior for the default, --strict_warnings, and --strict compiler modes (see *Setting the source language* on page 2-27 for details).

Example 1 - assigning the address of variable

Your code might contain constant expressions that assign the address of a variable, for example:

```
int i;
int j = (int)&i; /* but not allowed by ISO */
```

When compiling for C, this produces the following behavior:

*   In default mode the following warning is produced:

    ```
    Warning:  #1296-D: extended constant initialiser used
    ```

*   In --strict_warnings mode the following warning is produced:

    ```
    Warning:  #32-D: expression must have arithmetic type
    ```

*   In --strict mode, the following error is produced:

    ```
    Error:  #32: expression must have arithmetic type
    ```

Example 2 - constant value initializers

The compiler behavior when you have expressions that include constant values in C code is summarized in the following example:

```
                                    /* Std   RVCT v2.2 */extern int  const
c = 10;            /* ok       ok    */extern int  const x = c + 10;
/* error    ok    */static int      y = c + 10;       /* error      ok
*/static int  const z = c + 10;      /* error    ok      */extern int
*const cp = (int*)0x100;   /* ok        ok    */extern int *const xp = cp +
0x100;   /* error    ext   */static int *    yp = cp + 0x100;   /* error
ext    */static int *const zp = cp + 0x100;      /* error    ext    */
```

This indicates the behavior defined by the ISO C Standard (Std), and the behavior in RVCT:

*   ok indicates that the statement is accepted in all C modes.

*   ext is an extension to the ISO C Standard. The behavior depends on the strict mode used when compiling C:

    Nonstrict Accepted, without a warning.

    --strict_warnings
              Accepted, with the warning:

              ```
              Warning:  #1296-D: extended constant initialiser used
              ```

    --strict Conforms to the ISO C Standard, and gives the following error:

              ```
              Error:  #28: expression must have a constant value
              ```

See also *Pointer assignment and ROPI* on page 3-35 for an example that uses pointer assignment.

Array and pointer extensions

The following array and pointer extensions are supported:

- Assignment and pointer differences are permitted between pointers to types that are interchangeable but not identical, for example, **unsigned char** * and **char** *. This includes pointers to same-sized integral types, typically, **int** * and **long** *. A warning is issued, for example:

  ```
  #513-D: a value of type "char *" cannot be assigned to an entity of type
  "unsigned char *"
  ```
  Assignment of a string constant to a pointer to any kind of character is permitted without a warning.

- Assignment of pointer types is permitted in cases where the destination type has added type qualifiers that are not at the top level, for example, assigning **int** ** to **const int** **. Comparisons and pointer difference of such pairs of pointer types are also permitted. A warning is issued, for example:

  ```
  #42-D: operand types are incompatible ("const int **" and "int **")
  ```

- In operations on pointers, a pointer to void is always implicitly converted to another type if necessary. Also, a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO C, some operators permit these, and others do not.

- Pointers to different function types can be assigned or compared for equality (==) or inequality (!=) without an explicit type cast. A warning or error is issued, for example:

  ```
  #42-D: operand types are incompatible ("int (*)(char *)" and "unsigned int
  (*)(char *)")
  ```
  This extension is prohibited in C++ mode.

- A pointer to **void** can be implicitly converted to, or from, a pointer to a function type.

- In an initializer, a pointer constant value can be cast to an integral type if the integral type is big enough to contain it.

- A non-lvalue array expression is converted to a pointer to the first element of the array when it is subscripted or similarly used.

Preprocessor extensions

The following preprocessor extensions are supported:

- `#assert` preprocessing extensions of AT&T System V release 4 are permitted. These enable definition and testing of predicate names. Such names are in a namespace distinct from all other names, including macro names. A predicate name is given a definition by a preprocessing directive of the form:

  ```
  #assert name
  #assert name(token-sequence)
  ```

  In the first form, the predicate is not given a value. In the second form, it is given the value `token-sequence`.

  Such a predicate can be tested in a `#if` expression, as follows:

  ```
  #if name(token-sequence)
  ```

  This has the value 1 if a `#assert` of that name with that `token-sequence` has appeared, and 0 otherwise. A given predicate can be given more than one value at a given time.

  A predicate can be deleted by a preprocessing directive of the form:

  ```
  #unassert name
  #unassert name(token-sequence)
  ```

  The first form removes all definitions of the indicated predicate name. The second form removes only the indicated definition, leaving any other definitions.

- The nonstandard preprocessing directive `#include_next` is supported. This is a variant of the #include directive. It searches for the named file only in the directories on the search path that follow the directory where the current source file is found, that is, the one containing the `#include_next` directive. (This is an extension found in the GNU C compiler.)

Keywords

The ARM compiler implements some keyword extensions for types. See:

- *Type qualifiers* on page 3-26.

Other supported extensions

The following extensions are also supported:

- Static functions can be declared in function and block scopes. Their declarations are moved to the file-scope.

- A label definition can be followed immediately by a right brace. Normally, a statement must follow a label definition. The following warning is issued:

  ```
  #127-D: expected a statement
  ```

- An empty declaration, that is a semicolon with nothing before it, is permitted.

- The address of a variable with **register** storage class can be taken. The following warning is issued:

  ```
  #138-D: taking the address of a register variable is not allowed
  ```

- Benign redeclarations of **typedef** names are permitted. That is, a **typedef** name can be redeclared in the same scope as the same type. The following error is issued in strict ISO mode:

  ```
  #301: typedef name has already been declared (with same type)
  ```

- External entities declared in other scopes are visible. For example:

  ```
  void f1(void) { extern void f(); }
  void f2() { f(); } /* Using out of scope declaration */
  ```

  A warning is issued, for example:

  ```
  #676-D: using out-of-scope declaration of function "f" (declared at line 7)
  ```

- Numbers are scanned according to the syntax for numbers rather than the pp-number syntax. For example, 0x123e+1 is scanned as three tokens instead of one invalid token.

  —— **Note** ——

  If the --strict or --strict_warnings option is specified, then the pp-number syntax is used.

Compiler behavior that is undefined by the ISO C Standard

The following are considered undefined behavior by the ISO C Standard:

- Adjacent wide and non-wide string literals are not concatenated unless **wchar_t** and **char** are the same type. In C++ mode, when **wchar_t** is a keyword, adjacent wide and non-wide string literals are never concatenated.

- In character and string escapes, if the character following the \ has no special meaning, the value of the escape is the character itself, for example, \s is the same as s. The following warning is issued:

  ```
  #192-D: unrecognized character escape sequence
  ```

- A **struct** that has no named fields but at least one unnamed field is accepted by default, but the following error is issued in strict 1990 ISO Standard C mode:

  ```
  #64: declaration does not declare anything
  ```

3.2.3   C++ language extensions

The compiler supports the following extensions to the ISO/IEC C++ Standard:

*   *Inline and embedded assembler extensions*
*   *Read/write constants*
*   *Preprocessing* on page 3-46.

None of these extensions is available if the compiler is restricted to compiling strict ISO/IEC C++, for example, if you specify the `--strict` compiler option.

Inline and embedded assembler extensions

The following inline and embedded assembler extensions are supported:

*   `asm` statements and declarations are accepted in C++, see *Inline assembler* on page 4-2. The inline assembler is a way of extending the compiler, but there are limitations.

*   `asm` function declarations are accepted in C++, and inform the compiler that the function is an embedded assembler function, see *Embedded assembler* on page 4-20. `__asm` is a synonym for `asm` in C++. The embedded assembler gives a C++ interface to the ARM assembler. It provides greater flexibility than the inline assembler on the assembler instructions you can use, but only permits constant C++ expressions to be used and embedded `asm` functions cannot be inlined.

Read/write constants

For C++ only, a linkage specification for external constants indicates that a constant can be dynamically initialized or have mutable members.

——— **Note** ———

The use of `"C++:read/write"` linkage is only necessary for code compiled with `--apcs /rwpi`. If you recompile existing code with this option, you must change the linkage specification for external constants that are dynamically initialized or have mutable members.

Compiling C++ with the `--apcs /rwpi` option deviates from the ISO C++ Standard. The declarations in Example 3-5 on page 3-45 assume that x is in a read-only segment.

Example 3-5 External access

```
extern const T x;
extern "C++" const T x;
extern "C" const T x;
```

Dynamic initialization of x (including user-defined constructors) is not possible for the constants and T cannot contain mutable members. The new linkage specification in Example 3-6 declares that x is in a read/write segment (even if it was initialized with a constant). Dynamic initialization of x is permitted and T can contain mutable members. The definitions of x, y, and z in another file must have the same linkage specifications.

Example 3-6 Linkage specification

```
extern const int z;      /* in read-only segment, cannot  */
                         /* be dynamically initialized    */
extern "C++:read/write" const int y; /* in read/write segment */
                         /* can be dynamically initialized */
extern "C++:read/write" {
    const int i=5;            /* placed in read-only segment, */
                         /* not extern because implicitly static */
    extern const T x=6;   /* placed in read/write segment */
    struct S {
        static const T T x; /* placed in read/write segment */
    };
}
```

Constant objects must not be redeclared with another linkage. The code in Example 3-7 produces a compile error.

Example 3-7 Compiler error

```
extern "C++"  const T x;
extern "C++:read/write"  const T x; /* error */
```

———— **Note** ————

Because C does not have the linkage specifications, you cannot use a **const** object
declared in C++ as extern "C++:read/write" from C.

———————————————

Preprocessing

The preprocessing symbol c_plusplus is defined in addition to the standard __cplusplus
(see *Predefined macros* on page 3-89).

 ARM DUI 0282B

3.3   C and C++ implementation details

This section describes implementation details for the ARM compiler:

*   *Character sets and identifiers*
*   *Basic data types* on page 3-49
*   *Operations on basic data types* on page 3-51
*   *Structures, unions, enumerations, and bitfields* on page 3-53
*   *Variadic macros* on page 3-58
*   *Using the ::operator new function* on page 3-58
*   *Tentative arrays* on page 3-59
*   *Old-style C parameters in C++ functions* on page 3-59
*   *Anachronisms* on page 3-59
*   *Template instantiation* on page 3-60
*   *Namespaces* on page 3-61
*   *C++ exception handling* on page 3-63
*   *Extern inline functions* on page 3-64.

Compilation tools are restricted to the permitted endianness of the target MCU.

3.3.1   Character sets and identifiers

The following points apply to the character sets and identifiers expected by the compiler:

*   Uppercase and lowercase characters are distinct in all internal and external identifiers. An identifier can also contain a dollar (\$) character unless the `--strict` compiler option is specified. To permit dollar signs in identifiers with the `--strict` option, then also use the `--dollar` command-line option.

*   Calling `setlocale(LC_CTYPE, "ISO8859-1")` makes the `isupper()` and `islower()` functions behave as expected over the full 8-bit Latin-1 alphabet, rather than over the seven-bit ASCII subset. The locale must be selected at link-time. (See *Tailoring locale and CTYPE* on page 5-46.)

*   The characters in the source character set are assumed to be ISO 8859-1 (Latin-1 Alphabet), a superset of the ASCII character set. The printable characters are those in the range 32 to 126 and 160 to 255. Any printable character can appear in a string or character constant, and in a comment.

*   The ARM compiler supports multibyte character sets, such as Unicode.

*   Other properties of the source character set are host-specific.

The properties of the execution character set are target-specific. The ARM C and C++ libraries support the ISO 8859-1 (Latin-1 Alphabet) character set with the following consequences:

- The execution character set is identical to the source character set.

- There are eight bits in a character in the execution character set.

- There are four characters (bytes) in an **int**. If the memory system is:

    Little-endian   The bytes are ordered from least significant at the lowest address to most significant at the highest address.

    Big-endian   The bytes are ordered from least significant at the highest address to most significant at the lowest address. (On this product, only little-endian is supported.)

- In C all character constants have type **int**. In C++ a character constant containing one character has the type **char** and a character constant containing more than one character has the type **int**. Up to four characters of the constant are represented in the integer value. The last character in the constant occupies the lowest-order byte of the integer value. Up to three preceding characters are placed at higher-order bytes. Unused bytes are filled with the NULL (\0) character.

- All integer character constants that contain a single character, or character escape sequence (see Table 3-2), are represented in both the source and execution character sets.

Table 3-2 Character escape codes

Escape sequence	Char value	Description
\a	7	Attention (bell)
\b	8	Backspace
\t	9	Horizontal tab
\n	10	New line (line feed)
\v	11	Vertical tab
\f	12	Form feed
\r	13	Carriage return
\xnn	0xnn	ASCII code in hexadecimal
\nnn	0nnn	ASCII code in octal

- Characters of the source character set in string literals and character constants map identically into the execution character set.

- Data items of type **char** are unsigned by default. They can be explicitly declared as **signed char** or **unsigned char**:
  — the --signed_chars option can be used to make the **char** signed
  — the --unsigned_chars option can be used to make the **char** unsigned.

  ——— **Note** ———

  The --signed_chars option is not recommended for general use and is not required for ISO-compatible source. Code compiled with this option is not compliant with the *ABI for the ARM Architecture (base standard)* [BSABI], and incorrect use might result in a failure at runtime. This option is not supported by the C++ libraries.

  ————————————————

- No locale is used to convert multibyte characters into the corresponding wide characters (codes) for a wide character constant. This is not relevant to the generic implementation.

3.3.2 Basic data types

This section describes how the basic data types are implemented in ARM C and C++:
- *Size and alignment of basic data types*
- *Integer* on page 3-50
- *Float* on page 3-50
- *Arrays and pointers* on page 3-51.

Size and alignment of basic data types

Table 3-3 gives the size and natural alignment of the basic data types.

Table 3-3 Size and alignment of data types

Type	Size in bits	Natural alignment in bytes
char	8	1 (byte-aligned)
short	16	2 (halfword-aligned)
int	32	4 (word-aligned)
long	32	4 (word-aligned)
long long	64	8 (doubleword-aligned)

Table 3-3 Size and alignment of data types

Type	Size in bits	Natural alignment in bytes
`float`	32	4 (word-aligned)
`double`	64	8 (doubleword-aligned)
`long double`	64	8 (doubleword-aligned)
All pointers	32	4 (word-aligned)
`bool` (C++ only)	8	1 (byte-aligned)
`wchar_t` (C++ only)	16	2 (halfword-aligned)

Type alignment varies according to the context:

• Local variables are usually kept in registers, but when local variables spill onto the stack, they are always word-aligned. For example, a spilled local `char` variable has an alignment of 4.

• The natural alignment of a packed type is 1.

See *Structures, unions, enumerations, and bitfields* on page 3-53 for details.

——— **Note** ———

In RVCT v2.2, the compiler supports the option `--apcs /adsabi` to compile code that is compatible with the old ADS ABI. If you use this option, the alignment for `double` and `long long` types is 4. However, this is deprecated and will be removed in a future release.

Integer

Integers are represented in two's complement form. The low word of a `long long` is at the low address in little-endian mode, and at the high address in big-endian mode.

Float

Floating-point quantities are stored in IEEE format:
• `float` values are represented by IEEE single-precision values
• `double` and `long double` values are represented by IEEE double-precision values.

For `double` and `long double` quantities the word containing the sign, the exponent, and the most significant part of the mantissa is stored with the lower machine address in big-endian mode and at the higher address in little-endian mode. See *Operations on floating-point types* on page 3-52 for more information.

The ARM compiler implements an extension for floating-point constants (see *Hexadecimal floating-point constants* on page 3-35).

Arrays and pointers

The following statements apply to all pointers to objects in C and C++, except pointers to members:

*   Adjacent bytes have addresses that differ by one.
*   The macro NULL expands to the value 0.
*   Casting between integers and pointers results in no change of representation.
*   The compiler warns of casts between pointers to functions and pointers to data.
*   The type size_t is defined as unsigned int.
*   The type ptrdiff_t is defined as signed int.

3.3.3   Operations on basic data types

The ARM compiler performs the usual arithmetic conversions set out in relevant sections of the ISO C and C++ standards. The following sections document additional points that relate to arithmetic operations. See also *Statements* on page B-7.

Operations on integral types

The following statements apply to operations on the integral types:

*   All signed integer arithmetic uses a two's complement representation.

*   Bitwise operations on signed integral types follow the rules that arise naturally from two's complement representation. No sign extension takes place.

*   Right shifts on signed quantities are arithmetic.

*   Any quantity that specifies the amount of a shift is treated as an unsigned 8-bit value.

*   Any value to be shifted is treated as a 32-bit value.

*   Left shifts of more than 31 give a result of zero.

*   Right shifts of more than 31 give a result of zero from a shift of an unsigned value or positive signed value. They yield –1 from a shift of a negative signed value.

*   The remainder on integer division has the same sign as the divisor.

- If a value of integral type is truncated to a shorter signed integral type, the result is obtained by discarding an appropriate number of most significant bits. If the original number was too large, positive or negative, for the new type, there is no guarantee that the sign of the result is going to be the same as the original.

- A conversion between integral types does not raise an exception.

- Integer overflow does not raise an exception.

- Integer division by zero raises a **SIGFPE** exception. See Table 5-19 on page 5-115.

Operations on floating-point types

The following statements apply to operations on floating-point types:
- Normal IEEE 754 rules apply.
- Rounding is to the nearest representable value by default.
- Floating-point exceptions are disabled by default.

Also, see the description of the `--fpmode` option in *Defining optimization criteria* on page 2-43.

———— **Note** ————
The IEEE 754 standard for floating-point processing states that the default action to an exception is to proceed without a trap. You can modify floating-point error handling by tailoring the functions and definitions in `fenv.h`. See *Tailoring error signaling, error handling, and program exit* on page 5-71 for more details on floating-point processing.

Pointer subtraction

The following statements apply to all pointers in C. They also apply to pointers, other than pointers to members, in C++:

- When one pointer is subtracted from another, the difference is the result of the expression:

    ```
    ((int)a - (int)b) / (int)sizeof(type pointed to)
    ```
- If the pointers point to objects with a size of one, two, or four bytes, the natural alignment of the object ensures that the division is exact, provided the objects are not packed.

- For packed or longer types, such as **double** and **struct**, both pointers must point to elements of the same array.

3.3.4 Structures, unions, enumerations, and bitfields

This section describes the implementation of the structured data types **union**, **enum**, and **struct**. It also discusses structure padding and bitfield implementation.

For details of anonymous structures and unions, see *Anonymous classes, structures and unions* on page 3-36.

Unions

When a member of a **union** is accessed using a member of a different type, the resulting value can be predicted from the representation of the original type. No error is given.

Enumerations

An object of type **enum** is implemented in the smallest integral type that contains the range of the **enum**. The type of an **enum** is one of the following, according to the range of the **enum**:

- **unsigned char**
- **signed char**
- **unsigned short**
- **signed short**
- **unsigned int** (C++ always, C except when --strict is specified)
- **signed int**.

Implementing **enum** in this way can reduce data size. The command-line option --enum_is_int sets the underlying type of **enum** to **signed int**.

——— **Note** ———

The --enum_is_int option is not recommended for general use and is not required for ISO-compatible source. Code compiled with this option is not compliant with the *ABI for the ARM Architecture (base standard)* [BSABI], and incorrect use might result in a failure at runtime. This option is not supported by the C++ libraries.

Unless you use the --strict option, **enum** declarations can have a comma at the end as in:

```
enum { x = 1, };
```

Handling values that are out of range

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, a Warning is issued for out-of-range enumerator values:

```
#66: enumeration value is out of "int" range
```

Such values are treated the same way as in C++, that is, they are treated as `unsigned int`, `long long`, or `unsigned long long`.

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

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

Structures

The following points apply to:
* all C structures
* all C++ structures and classes not using virtual functions or base classes.

Structure alignment

> The alignment of a non-packed structure is the maximum alignment required by any of its fields.

Field alignment

> Structures are arranged with the first-named component at the lowest address. Fields are aligned as follows:
> * A field with a `char` type is aligned to the next available byte.
> * A field with a `short` type is aligned to the next even-addressed byte.
> * Bitfield alignment depends on how the bitfield is declared. See *Bitfields in packed structures* on page 3-58 for more information.
> * All other types are aligned on word boundaries.

Structures can contain padding to ensure that fields are correctly aligned and that the structure itself is correctly aligned. on page 3-55 shows an example of a conventional, non-packed structure. Bytes 1, 2, and 3 are padded to ensure correct field alignment. Bytes 11 and 12 are padded to ensure correct structure alignment. The `sizeof()` function returns the size of the structure including padding.

```
struct {char c; int x; short s} ex1;
```


Figure 3-1 Conventional structure example

The compiler pads structures in one of the following ways, according to how the structure is defined:

• Structures that are defined as **static** or **extern** are padded with zeros.

• Structures on the stack or heap, such as those defined with malloc() or **auto**, are padded with whatever was previously stored in those memory locations. You cannot use memcmp() to compare padded structures defined in this way (see Figure 3-1).

Use the --remarks option to view the messages that are generated when the compiler inserts padding in a **struct**.

Structures with empty initializers are permitted in C++:

```
struct { int x; } X = { };
```

However, if you are compiling C, or compiling C++ with the --c90 options, an error is generated, for example:

```
Error:  #29: expected an expression
```

Packed structures

A packed structure is one where the alignment of the structure, and of the fields within it, is always 1.

Packed structures are defined with the __packed qualifier, see *Type qualifiers* on page 3-26. There is no command-line option to change the default packing of structures.

Bitfields

In non-packed structures, the ARM compiler allocates bitfields in *containers*. A container is a correctly aligned object of a declared type.

Bitfields are allocated so that the first field specified occupies the lowest-addressed bits of the word, depending on configuration:

Little-endian   Lowest addressed means least significant.

Big-endian   Lowest addressed means most significant. (On this product, only little-endian is supported.)

A bitfield container can be any of the integral types.

————— **Note** —————

In strict 1990 ISO Standard C, the only types permitted for a bit field are `int`, `signed int` and `unsigned int`. For non `int` bitfields, the compiler displays the following error:

```
Error:  #230: nonstandard type for a bit field
```

A plain bitfield, declared without either `signed` or `unsigned` qualifiers, is treated as `unsigned`. For example, int x:10 allocates an unsigned integer of 10 bits.

A bitfield is allocated to the first container of the correct type that has a sufficient number of unallocated bits, for example:

```
struct X {
    int x:10;
    int y:20;
};
```

The first declaration creates an integer container and allocates 10 bits to x. At the second declaration, the compiler finds the existing integer container with a sufficient number of unallocated bits, and allocates y in the same container as x.

A bitfield is wholly contained within its container. A bitfield that does not fit in a container is placed in the next container of the same type. For example, the declaration of z overflows the container if an additional bitfield is declared for the structure:

```
struct X {
    int x:10;
    int y:20;
    int z:5;
};
```

The compiler pads the remaining two bits for the first container and assigns a new integer container for z.

Bitfield containers can *overlap* each other, for example:

```
struct X {
    int x:10;
    char y:2;
};
```

The first declaration creates an integer container and allocates 10 bits to x. These 10 bits occupy the first byte and two bits of the second byte of the integer container. At the second declaration, the compiler checks for a container of type **char**. There is no suitable container, so the compiler allocates a new correctly aligned **char** container.

Because the natural alignment of **char** is 1, the compiler searches for the first byte that contains a sufficient number of unallocated bits to completely contain the bitfield. In the example structure, the second byte of the **int** container has two bits allocated to x, and six bits unallocated. The compiler allocates a **char** container starting at the second byte of the previous **int** container, skips the first two bits that are allocated to x, and allocates two bits to y.

If y is declared char y:8, the compiler pads the second byte and allocates a new **char** container to the third byte, because the bitfield cannot overflow its container. The bitfield allocation for the following example structure is shown in Figure 3-2:

```
struct X {
    int x:10;
    char y:8;
};
```

Bit number			
31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
unallocated	y	padding	x

Figure 3-2 Bitfield allocation 1

——— **Note** ———

The same basic rules apply to bitfield declarations with different container types. For example, adding an **int** bitfield to the example structure gives:

```
struct X {
    int x:10;
    char y:8;
    int z:5;
}
```

The compiler allocates an **int** container starting at the same location as the int x:10 container and allocates a byte-aligned **char** and 5-bit bitfield (see Figure 3-3).

Bit number				
31 30 29	28 27 26 25 24 23 22 21 20	19 18 17 16 15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
free	z	y	padding	x

Figure 3-3 Bitfield allocation 2

You can explicitly pad a bitfield container by declaring an unnamed bitfield of size zero. A bitfield of zero size fills the container up to the end if the container is not empty. A subsequent bitfield declaration starts a new empty container.

Bitfields in packed structures

Bitfield containers in packed structures have an alignment of 1. Therefore, the maximum bit padding for a bitfield in a packed structure is 7 bits. For an unpacked structure, the maximum padding is 8*sizeof(container-type)-1 bits.

3.3.5 Variadic macros

Variadic macros are supported, for example:

```
#define eprintf(...) fprintf(stderr, __VA_ARGS__)
```

3.3.6 Using the ::operator new function

In RVCT v2.x, when the ::operator new function runs out of memory, it raises the signal **SIGOUTOFHEAP**, instead of throwing a C++ exception. See *Signal functions* on page 5-115 for more details.

In RVCT v2.x ::operator new(std::size_t) throws when memory allocation fails, in accordance with the ISO C++ Standard, rather than raising a signal. If the exception is not caught, std::terminate() is called. The compiler option --force_new_nothrow turns all new calls in a compilation into calls to ::operator new(std::size_t, std::nothrow_t&) or :operator new[](std::size_t, std::nothrow_t&). However, this does not affect operator new calls in libraries, nor calls to any class-specific operator new. It is possible to install a new_handler which raises a signal, to restore the RVCT v2.x behavior.

See *C++ Language configuration and object generation* on page 2-32 for a description of the --force_new_nothrow option.

3.3.7 Tentative arrays

The ADS v1.2 and RVCT v1.2 C++ compilers enabled you to use incomplete (tentative) array declarations, for example, `int a[]`. You cannot use tentative arrays when compiling C++ with the RVCT v2.x compilers.

3.3.8 Old-style C parameters in C++ functions

The ADS v1.2 and RVCT v1.2 C++ compilers enabled you to use old-style C parameters in C++ functions. That is,

```
void f(x) int x; { }
```

In the RVCT v2.x compilers, you must use the `--anachronisms` compiler option if your code contains any old-style parameters in functions. The compiler warns you if it finds any instances.

3.3.9 Anachronisms

The following anachronisms are accepted when you enable anachronisms with the `--anachronisms` option (see *Setting the source language* on page 2-27):

- **overload** is permitted in function declarations. It is accepted and ignored.

- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes, because these must always be defined.

- The number of elements in an array can be specified in an array delete operation. The value is ignored.

- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.

- The base class name can be omitted in a base class initializer if there is only one immediate base class.

- Assignment to the `this` pointer in constructors and destructors is permitted.

- A bound function pointer, that is, a pointer to a member function for a given object, can be cast to a pointer to a function.

- A nested class name can be used as a non-nested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.

- A reference to a non-const type can be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.

- A reference to a non-const class type can be initialized from an rvalue of the class type or a class derived from that class type. No, additional, temporary is used.

- A function with old-style parameter declarations is permitted and can participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following declares the overloading of two functions named f:

```
int f(int);
int f(x) char x; { return x; }
```

——— **Note** ———

In C, this code is legal but has a different meaning. A tentative declaration of f is followed by its definition.

3.3.10   Template instantiation

The ARM compiler does all template instantiations automatically, and makes sure there is only one definition of each template entity left after linking. The compiler does this by emitting template entities in named common sections. Therefore, all duplicate common sections, that is, common sections with the same name, are eliminated by the linker.

——— **Note** ———

You can limit the number of concurrent instantiations of a given template with the --pending_instantiations compiler option.

See *C++ Language configuration and object generation* on page 2-32 for details of the compiler options associated with templates.

Implicit inclusion

When implicit inclusion is enabled, the compiler assumes that if it requires a definition to instantiate a template entity declared in a .h file it can implicitly include the corresponding .cc file to get the source code for the definition. For example, if a template entity ABC::f is declared in file xyz.h, and an instantiation of ABC::f is required

in a compilation but no definition of ABC::f appears in the source code processed by the compilation, then the compiler checks to see if a file xyz.cc exists, If this file exists, the compiler processes the file as if it were included at the end of the main source file.

To find the template definition file for a given template entity the compiler has to know the full path name of the file where the template was declared and whether the file was included using the system include syntax (for example, #include <file.h>). This information is not available for preprocessed source containing #line directives. Consequently, the compiler does not attempt implicit inclusion for source code containing #line directives.

The compiler looks for the definition-file suffixes .cc and .CC.

You can turn implicit inclusion mode on or off with the --implicit_include (the default) and --no_implicit_include command-line options. See *C++ Language configuration and object generation* on page 2-32 for details of these options.

Implicit inclusions are only performed during the normal compilation of a file, that is, when not using the -E command-line option (see *Setting preprocessor options* on page 2-31).

3.3.11   Namespaces

When doing name lookup in a template instantiation, some names must be found in the context of the template definition. Other names can be found in the context of the template instantiation. The compiler implements two different instantiation lookup algorithms:

•   the algorithm mandated by the standard, and referred to as dependent name lookup (see *Dependent name lookup processing*)

•   the algorithm that existed before dependent name lookup was implemented.

Dependent name lookup is done in strict mode, unless explicitly disabled by another command-line option, or when dependent name processing is enabled by either a configuration flag or a command-line option.

See *C++ Language configuration and object generation* on page 2-32 for details of the options associated with namespaces.

Dependent name lookup processing

When doing dependent name lookup, the compiler implements the instantiation name lookup rules specified in the standard. This processing requires that non-class prototype instantiations be done. This in turn requires that the code be written using the typename and template keywords as required by the standard.

Lookup using the referencing context

When not using dependent name lookup, the compiler uses a name lookup algorithm that approximates the two-phase lookup rule of the standard, but in a way that is more compatible with existing code and existing compilers.

When a name is looked up as part of a template instantiation, but is not found in the local context of the instantiation, it is looked up in a synthesized instantiation context. This synthesized instantiation context includes both names from the context of the template definition and names from the context of the instantiation. For example:

```
namespace N {
    int g(int);
    int x = 0;
    template <class T> struct A {
        T f(T t) { return g(t); }
        T f() { return x; }
    };
}namespace M {
    int x = 99;
    double g(double);
    N::A<int> ai;
    int i = ai.f(0);        // N::A<int>::f(int) calls N::g(int)
    int i2 = ai.f();        // N::A<int>::f() returns 0 (= N::x)
    N::A<double> ad;
    double d = ad.f(0);     // N::A<double>::f(double) calls M::g(double)
    double d2 = ad.f();     // N::A<double>::f() also returns 0 (= N::x)
}
```

The lookup of names in template instantiations does not conform to the rules in the standard in the following respects:

*   Although only names from the template definition context are considered for names that are not functions, the lookup is not limited to those names visible at the point where the template was defined.

*   Functions from the context where the template was referenced are considered for all function calls in the template. Functions from the referencing context are only visible for dependent function calls.

Argument-dependent lookup

When argument-dependent lookup is enabled, functions that are made visible using argument-dependent lookup can overload with those made visible by normal lookup. The standard requires that this overloading occur even when the name found by normal

lookup is a block extern declaration. The compiler does this overloading, but in default mode, argument-dependent lookup is suppressed when the normal lookup finds a block extern.

This means a program can have different behavior, depending on whether it is compiled with or without argument-dependent lookup, even if the program makes no use of namespaces. For example:

```
struct A { };
A operator+(A, double);
void f() {
    A a1;
    A operator+(A, int);
    a1 + 1.0;          // calls operator+(A, double) with arg-dependent lookup
}                      // enabled but otherwise calls operator+(A, int);
```

3.3.12   C++ exception handling

C++ exception handling is fully supported in RVCT. However, the compiler does not support this by default. You must enable C++ exception handling with the --exceptions option (see *C++ Language configuration and object generation* on page 2-32).

———— **Note** ————

The Rogue Wave Standard C++ Library is provided with C++ exceptions enabled.

You can exercise limited control over exception table generation (see *Function unwinding at runtime*).

Function unwinding at runtime

By default, functions compiled with --exceptions can be unwound at runtime (see *C++ Language configuration and object generation* on page 2-32). *Function unwinding* includes destroying C++ automatic variables, and restoring register values saved in the stack frame. Function unwinding is implemented by emitting an exception table describing the operations to be performed.

You can enable or disable unwinding for specific functions with the pragmas #pragma exceptions_unwind and #pragma no_exceptions_unwind (see *Pragmas controlling code generation* on page 3-5). The --exceptions_unwind option sets the initial value of this pragma.

Disabling function unwinding for a function has the following effects:

• Exceptions cannot be thrown through that function at runtime, and no stack unwinding occurs for that throw. If the throwing language is C++, then `std::terminate` is called.

• A very compact exception table representation can be used to describe the function, which assists smart linkers with table optimization.

• Function inlining is restricted, because the caller and callee must interact correctly.

Therefore, #pragma `no_exceptions_unwind` can be used to forcibly prevent unwinding in a way that requires no additional source decoration.

By contrast, in C++ an empty function exception specification permits unwinding as far as the protected function, then calls `std::unexpected()` in accordance with the ISO C++ Standard.

3.3.13 Extern inline functions

The ISO C++ Standard requires inline functions to be defined wherever you use them. To prevent the clashing of multiple out-of-line copies of inline functions, the ARM C++ compiler emits out-of-line `extern` functions in common sections.

Out-of-line inline functions

The compiler emits inline functions out-of-line, in the following cases:

• The address of the function is taken, for example:

```
inline int g() {return 1;}
int (*fp)() = &g
```

• The function cannot be inlined, for example, a recursive function:

```
inline int g() {return g();}
```

• The heuristic used by the compiler decides that it is better not to inline the function. This heuristic is influenced by -Ospace and -Otime. If you use -Otime, the compiler inlines more functions. You can override this heuristic by declaring a function with `__forceinline` (see *Function storage class modifiers* on page 3-13), for example:

```
__forceinline int g() {return 1;}
```

You can also use the --forceinline compiler option (see *Defining optimization criteria* on page 2-43).

3.4 GNU extensions to the ARM compiler

This section describes the GNU compiler extensions that are supported by the ARM compiler. These extensions are supported 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 in this document as:

ARM mode   The default mode. That is, compilations without the `--gnu` option.

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

Table 3-4 lists the GNU extensions that are supported by the ARM compiler, and the mode and language in which they are supported.

Table 3-4 Supported GNU extensions

GNU extension	Supported mode	Supported language
Alignment on page 3-68	GNU, ARM	C, C++
Alternate keywords on page 3-68	GNU	C, C++
Assembler labels on page 3-68	GNU	C, C++
Attribute syntax on page 3-68	GNU, ARM[a]	C, C++
Builtin functions on page 3-69	GNU	C, C++
C++ comments on page 3-70	GNU, ARM	C, C++
Case ranges on page 3-70	GNU	C, C++
Cast of a union on page 3-70	GNU	C only
Character escapes on page 3-71	GNU	C, C++
Compound literals on page 3-71	GNU	C only
Conditionals on page 3-71	GNU	C only
Designated inits on page 3-72	GNU	C, C++
Dollar signs on page 3-72	GNU	C, C++
Function attributes on page 3-72	GNU, ARM	C, C++
Function names on page 3-75	GNU, ARM	C only
Function prototypes on page 3-76	GNU	C only

Table 3-4 Supported GNU extensions

GNU extension	Supported mode	Supported language
Hex floats on page 3-76	GNU	C, C++
Incomplete enums on page 3-77	GNU, ARM	C, C++
Initializers on page 3-77	GNU	C, C++
Inline on page 3-77	GNU, ARM	C, C++
Local labels on page 3-78	GNU	C, C++
Long long on page 3-78	GNU, ARM	C, C++
Lvalues on page 3-79	GNU	C, C++
Multiline strings on page 3-79	GNU	C, C++
Pointer arithmetic on page 3-80	GNU	C, C++
Return and frame addresses on page 3-80	GNU	C, C++
Statement expressions on page 3-81	GNU	C, C++
Subscripting struct on page 3-81	GNU, ARM	C, C++
Type attributes on page 3-82	GNU, ARMa	C, C++
typeof on page 3-83	GNU	C, C++
Unnamed fields on page 3-83	GNU	C, C++
Variable attributes on page 3-83	GNU, ARMa	C, C++
Variadic macros on page 3-88	GNU, ARM	C, C++
Zero length arrays on page 3-88	GNU	C, C++

a. the `transparent_union` attribute is only supported in GNU mode

―――― **Note** ――――

For more details on the use of these extensions, see the GNU compiler documentation. You can access this documentation online at `http://gcc.gnu.org`.

3.4.1   Alignment

You can use the keyword **__alignof__** to inquire about the alignment of a type or variable, for example:

```
int Alignment_0()
{
    return __alignof__ (int);
}
```

Supported in GNU mode and ARM mode. Also, see *__ALIGNOF__* on page 3-14.

3.4.2   Alternate keywords

The compiler recognizes alternate keywords of the form __*keyword*__. These alternate keywords have the same behavior as the original keywords. For example, **__const__** behaves the same as **const**.

Supported in GNU mode only.

3.4.3   Assembler labels

Specifies the assembler name to use for a C symbol. For example, you might have assembler code and C code that uses the same symbol name, such as counter. Therefore, you can export a different name to be used by the assembler:

```
int counter __asm__("counter_v1") = 0;
```

This exports the symbol counter_v1 and not the symbol counter.

Supported in GNU mode only.

3.4.4   Attribute syntax

The keyword __attribute__ enables you to specify special attributes of variables or structure fields, functions, and types. The keyword format is either:

```
__attribute__ (attribute)
__attribute__ (__attribute__)
```

For example:

```
void * Function_Attributes_malloc_0(int b) __attribute__ ((malloc));
int b __attribute__ ((__packed__));
```

Unless noted otherwise, this keyword is supported in GNU mode and ARM mode, and if you specify the --strict option.

See the following sections for details of the attributes that are supported:
- *Function attributes* on page 3-72
- *Type attributes* on page 3-82
- *Variable attributes* on page 3-83.

3.4.5   Builtin functions

This section describes the GNU builtin functions that are supported:
- *__builtin_constant_p function*
- *__builtin_frame_address*
- *__builtin_return_address* on page 3-70.

__builtin_constant_p function

You can use the builtin function __builtin_constant_p to determine if a value is known to be constant at compile-time. The argument of the function is the value to test. The function returns the integer 1 if the argument is known to be a compile-time constant and 0 if it is not known to be a compile-time constant. A return of 0 does not indicate that the value is not a constant, but that the compiler cannot prove it is a constant with the specified value of the -0 option.

```
int Other_Builtins_2(int a)
{
    const int aconst =10;
    int notconst =20;
    static int aresult ;
    static int notresult;
    aresult= aconst;
    aresult += notconst;
    aresult = __builtin_constant_p (aconst);
    notresult = __builtin_constant_p (notconst);
    return (aresult+notresult);
}
```

Supported in GNU mode only.

__builtin_frame_address

See *Return and frame addresses* on page 3-80 for details.

__builtin_return_address

See *Return and frame addresses* on page 3-80 for details.

3.4.6 C++ comments

You can use // to indicate the start of a one-line comment.

Supported in GNU mode and ARM mode.

3.4.7 Case ranges

You can specify ranges of values in switch statements, for example:

```
int Case_Ranges_0(int arg)
{
    int aLocal;
    int bLocal =arg;
    switch (bLocal)
    {
    case 0 ... 10:
        aLocal= 1;
        break;
    case 11 ... 100:
        aLocal =2;
        break;
    default:
        aLocal=-1;
    }
    return aLocal;
}
```

Supported in GNU mode only.

3.4.8 Cast of a union

A cast to a union type is similar to other casts, except that the type specified is a union type. You can specify the type either with a union tag or with a typedef name.

```
typedef union { double d; int i;  }foo_t;
int Cast_to_Union_0(int a, double b)
{

    foo_t u;
    if (a>100)
        u = (foo_t) a ; // automatically equivalent to u.i=a;
```

```
        else
            u = (foo_t) b ; // automatically equivalent to u.d=b;
        return u.i;
    }
```

Supported in GNU mode for C only.

3.4.9   Character escapes

In strings, the escape sequence '\e' stands for <ESC> (ASCII 27). Compare this with the use of '\n' for newline (ASCII 10).

Supported in GNU mode only.

3.4.10   Compound literals

ISO C99 supports compound literals. A compound literal looks like a cast followed by an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer. It is an lvalue.

For example:

```
int y[] = (int []) {1, 2, 3};
int z[] = (int [3]) {1};
```

Supported in GNU mode for C only.

3.4.11   Conditionals

The middle operand in a conditional statement can be omitted, if the result is to be the same as the test, for example:

```
i ? i : j
```

This is most useful if the test modifies the value in some way, for example:

```
i++ ? : j
```

where i++ comes from a macro. If you write code in this way, then i++ is evaluated only once.

Supported in GNU mode for C only.

3.4.12   Designated inits

Enables you to label the elements of initializers, so that items can be listed out of order or with gaps, for example:

```
int a[6] = { [4] = 29, [2] = 15 };
int b[6] = { 0,0,15,0,29,0}; // a[] is equivalent to b[]
```

Supported in GNU mode, in both C and C++.

3.4.13   Dollar signs

The dollar sign ($) is permitted in identifiers.

Supported in GNU mode and ARM mode (see *Identifiers* on page 3-31). Also, the --dollar option is enabled by default (see *Controlling implementation details* on page 2-54), and is supported in both modes.

3.4.14   Function attributes

This section describes the function attributes that are supported:
- *const*
- *deprecated* on page 3-73
- *malloc* on page 3-73
- *noreturn* on page 3-73
- *pure* on page 3-74
- *section* on page 3-74
- *unused* on page 3-75
- *weak* on page 3-75.

const

Many functions examine only the arguments passed to them, and have no effects except for the return value. This is a more strict class than __attribute__((pure)) (see *pure* on page 3-74), because a function is not permitted to read global memory. If a function is known to operate only on its arguments then it can be subject to common subexpression elimination and loop optimizations.

```
int Function_Attributes_const_0(int b) __attribute__ ((const));
int Function_Attributes_const_2(int b)
{
    int aLocal=0;
    aLocal += Function_Attributes_const_0(b);
```

```
    aLocal += Function_Attributes_const_0(b);
    return aLocal;
}
```

In this code `Function_Attributes_const_0` might be called once only, and the result is doubled to get the correct return value.

Supported in GNU mode and ARM mode, and if you specify the `--strict` option.

deprecated

The `deprecated` function attribute indicates that a function exists but the compiler must generate a warning if the deprecated function is used.

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

Supported in GNU mode only, and if you specify the `--strict` option.

malloc

The `malloc` function attribute indicates that the function can be treated like `malloc` and the associated optimizations can be performed.

```
void * Function_Attributes_malloc_0(int b) __attribute__ ((malloc));
```

Supported in GNU mode only, and if you specify the `--strict` option.

noreturn

The `noreturn` function attribute informs the compiler that the function does not return. The compiler can then perform optimizations by removing the code that is never reached.

```
int Function_Attributes_NoReturn_0(void) __attribute__ ((noreturn));
```

Supported in GNU mode only, and if you specify the `--strict` option.

The ARM mode equivalent (see *Function qualifiers* on page 3-9) is `__declspec(noreturn)`, for example:

```
__declspec(noreturn) int Function_Attributes_NoReturn_0(void);
```

pure

Many functions have no effects except to return a value, and that the return value depends only on the parameters and global variables. Functions of this kind can be subject to data flow analysis and might be eliminated.

```
int Function_Attributes_pure_0(int b) __attribute__ ((pure));
int Function_Attributes_pure_0(int b)
{
    static int aStatic;
    aStatic++;
    return b++;
}

int Function_Attributes_pure_2(int b)
{
    int aLocal=0;
    aLocal += Function_Attributes_pure_0(b);
    return 0;
}
```

In this example, the call to `Function_Attributes_pure_0` might be eliminated because its result is not used.

Supported in GNU mode only, and if you specify the `--strict` option.

Using `__attribute__((pure))` is equivalent to using `__pure` (see *Function qualifiers* on page 3-9). `__pure` is supported in GNU Mode as well as ARM mode.

section

The `section` function attribute enables you to place code in different sections of the image.

```
void Function_Attributes_section_0 (void) __attribute__ ((section
("new_section")));
void Function_Attributes_section_0 (void)
{
    static int aStatic =0;
    aStatic++;
}
```

In this example, `Function_Attributes_section_0` is placed into the RO section `new_section` rather than `.text`.

Supported in GNU mode only, and if you specify the `--strict` option.

unused

The unused function attribute prevents the compiler from generating warnings if the function is not referenced. This does not change the behavior of the unused function removal process.

```
static int Function_Attributes_unused_0(int b) __attribute__ ((unused));
```

Supported in GNU mode only, and if you specify the `--strict` option.

weak

Functions defined with `__attribute__((weak))` export their symbols weakly. A weakly defined function behaves like a normally defined function unless a non-weakly defined function of the same name is linked into the same image.If both a non-weakly defined function and a weakly defined function exist in the same image then all calls to the function resolve to call the non-weak function. If multiple weak definitions are available, the linker chooses one for use by all calls.

Weakly defined functions cannot be inlined.

Unlike the `__weak` keyword, functions declared with `__attribute__((weak))` and then defined without `__attribute__((weak))` behave as *weak* functions (see *Keywords specific to the ARM compiler* on page 3-22 for details).

```
extern int Function_Attributes_weak_0 (int b) __attribute__ ((weak));
```

Supported in GNU mode and ARM mode, and if you specify the `--strict` option.

3.4.15   Function names

Two predefined macros are supported to hold the name of the current function:

`__FUNCTION__`   This holds the name of the function as it appears in the source.

`__PRETTY_FUNCTION__`

This holds the name of the function pretty printed in a language-specific fashion.

For example:

```
void Function_Names_0()
{
    printf("Normal is %s. Pretty is %s\n", __FUNCTION__,__PRETTY_FUNCTION__);
}
```

Supported in GNU mode and ARM mode.

Also, see *Predefined macros* on page 3-89.

3.4.16 Function prototypes

The compiler recognizes function prototypes that override old-style non-prototype definitions that appear at a later position in your code. For example:

```
int Function_Prototypes_0 (char);
// Old-style function definition.
int Function_Prototypes_0 (x)
    char x;
{
    return x == 0;
}
```

The ARM compiler generates a warning message, for example:

```
Warning:  #1294-D: Old-style function Function_Prototypes_0
```

Supported in GNU mode, for C only.

3.4.17 Hex floats

ISO C99 supports floating-point numbers that can be written in the usual decimal notation, such as `1.55e1`, but also numbers that can be written in hexadecimal format, for example, `0x1.fp3`. In that format the `0x` hexadecimal introducer and the `p` or `P` exponent field are mandatory. The exponent is a decimal number that indicates the power of 2 by which the significant part will be multiplied. Therefore, `0x1.f` is 1 15/16, `p3` multiplies it by 8, and the value of `0x1.fp3` is the same as `1.55e1`.

```
float Hex_Floats_0()
{
    return 0x1.fp3;
}
```

Supported in GNU mode. However, to support hex floats you must specify the pragma `#pragma import(__use_c99_library)`.

Supported in ARM mode when not using the option `--strict` (see *Setting the source language* on page 2-27).

3.4.18   Incomplete enums

You can specify forward declarations of enums, for example:

```
enum Incomplete_Enums_0;
int Incomplete_Enums_2 (enum Incomplete_Enums_0 * passon)
{
    return 0;
}
int Incomplete_Enums_1 (enum Incomplete_Enums_0 * passon)
{
    return Incomplete_Enums_2(passon);
}
enum Incomplete_Enums_0 { ALPHA, BETA, GAMMA };
```

Supported in GNU mode and ARM mode, for C and C++.

3.4.19   Initializers

As in standard C++ and ISO C99, the elements of an aggregate initializer for an automatic variable are not required to be constant expressions, for example:

```
float Initializers_0 (float f, float g)
{
    float beat_freqs[2] = { f-g, f+g };
    float aLocal;
    int i=0;
    for (;i<2;i++)
        aLocal+=beat_freqs[i];
    return aLocal;
}
```

Supported in GNU mode only.

3.4.20   Inline

The inline function qualifier specifies that the function is to be inlined.

static inline foo (){...}

> foo is used internally to the file, and the symbol is not exported.

inline foo(){...}

> foo is used internally to the file and an out of line version is made available and the name foo exported.

```
extern inline foo (){...}
```

> In GNU mode, foo is used internally if it is inlined. If it is not inlined then an external version is referenced rather than using a call to the internal version. Also, the foo symbol is not emitted.
>
> In ARM mode, extern is ignored and the functionality is the same as inline foo() for C++. In ARM C, you must use __inline. See *Extern inline functions* on page 3-64 for more details.

3.4.21 Local labels

Labels that are local to a statement-expression (see *Statement expressions* on page 3-81) are useful for local labels in macros, where the macro might be included multiple times in the same function. For example:

```
({
__label__ found;
...
goto found;
...
found:;
})
```

Supported in GNU mode only.

3.4.22 Long long

ISO C99 supports data types for integers that are at least 64 bits wide:

- for a signed integer, write long long int

- for an unsigned integer, write unsigned long long int

- to make an integer constant of type long long int, add the suffix LL to the integer

- to make an integer constant of type unsigned long long int, add the suffix ULL to the integer.

Supported in GNU mode and ARM mode (see *long long* on page 3-33).

3.4.23   Lvalues

GNU mode enables a more relaxed definition on what constitutes an lvalue when looking at comma expressions and ?: constructs. You can use compound expressions, conditional expressions, and casts as follows:

*   You can assign a compound expression:

    ```
    (a, b) += x;
    ```
    This is equivalent to:

    ```
    temp = (a,b);
    b = temp + x
    ```
    This causes the ARM compiler to generate a warning, for example:

    ```
    Warning:  #174-D: expression has no effect
    ```

*   You can get the address of a compound expression &(a, b). This is equivalent to (a, &b).

*   You can use conditional expressions, for example:

    ```
    (a ? b : c) = a;
    ```
    This picks b or c as the destination dependent on a.

Supported in GNU mode only.

3.4.24   Multiline strings

String literals can include newline characters in addition to the escaped counterpart ('\n'), for example:

```
void Multi_line_Strings_0()
{
    printf (" this is a string with a line break
here between the words ");
}
```

———— **Note** ————

The ARM compiler supports this behavior in --gnu mode for backwards compatibility. It is no longer supported in the GNU compiler.

—————————————————

3.4.25 Pointer arithmetic

You can perform arithmetic on void pointers and function pointers. Also sizeof a void type or a function type is defined to be 1. For example:

```
int Pointer_Arith_0()
{
    void * pointer;
    return sizeof *pointer;
}
int Pointer_Arith_1()
{
    static ptrdiff_t aLocal;
    aLocal= Pointer_Arith_0 - Pointer_Arith_1;
    return sizeof Pointer_Arith_0;
}
```

The ARM compiler generates a warning when it detects pointer arithmetic, for example:

```
Warning:  #1254-D: arithmetic on pointer to void or function type
```

Supported in GNU mode, in both C and C++.

3.4.26 Return and frame addresses

This extension enables you to use the following builtin functions:

`__builtin_frame_address`

Obtains the frame address of a function.

`__builtin_return_address`

Obtains the return address of a function.

The argument to `__builtin_frame_address` and `__builtin_return_address` must be zero.

The following example shows how to use these builtin functions:

```
int Return_Address_0 ()
{
    static void *aLocal;
    static void *bLocal;
    aLocal = __builtin_return_address (0); // should return the lr
    bLocal = __builtin_frame_address (0);  // should return the sp
    return aLocal + bLocal;
}
```

Supported in GNU mode only.

Also, see the ARM mode equivalent *__return_address() intrinsic* on page 3-16.

3.4.27   Statement expressions

Statement expressions enable you to place whole sections of code, including declarations, within ({ }) braces. The result of statement expressions is the final item in the statement list.

```
if (({ int y = foo;
    int z;
    if (y > 0) z = y;
    else z = - y;
    z>b; }))
    return b;
```

This gives you more powerful macro expansion options and is used to inline code sections. For example:

```
#define __get_user_asm_half(x,addr,err)\
({                                      \
    unsigned long __b1, __b2, __ptr = (unsigned long)addr;\
    __get_user_asm_byte(__b1, __ptr, err);\
    __get_user_asm_byte(__b2, __ptr + 1, err);\
    (x) = __b1 | (__b2 << 8);\
})
```

Supported in GNU mode only.

3.4.28   Subscripting struct

In ISO C99, arrays that are not lvalues still decay to pointers, and can be subscripted. However, you must not modify or use them after the next sequence point, and you must not apply the unary & operator to them. Arrays of this kind can be subscripted in C89 mode, but they do not decay to pointers outside C99 mode.

```
struct Subscripting_Struct {int a[4];};
extern struct Subscripting_Struct Subscripting_0(void);
int Subscripting_1 (int index)
{
    return Subscripting_0().a[index];
}
```

Supported in GNU mode and ARM mode.

3.4.29 Type attributes

This section describes the type attributes that are supported:

- *aligned*
- *packed*
- *transparent_union*.

aligned

The `aligned` type attribute specifies a minimum alignment for the type.

Supported in GNU mode and ARM mode, and if you specify the `--strict` option.

packed

The `packed` type attribute specifies that a type must have the smallest possible alignment.

The compiler displays the following warning if you use this attribute in a `typedef`:

```
Warning:  #1210-D: the "packed" attribute is ignored in a typedef
```

Supported in GNU mode and ARM mode, and if you specify the `--strict` option.

transparent_union

The `transparent_union` type attribute enables you to set up a `transparent_union` type, for example:

```
typedef union
{
    int myint;
    float myfloat;
}Type_Attributes_transparent_union_t __attribute__ ((transparent_union));
void Type_Attributes_transparent_union_0(
        Type_Attributes_transparent_union_t aUnion)
{
    static int aStatic;
    aStatic +=aUnion.myint;
}
void Type_Attributes_transparent_union_1()
{
    int aLocal =0;
    float bLocal =0;
```

```
            Type_Attributes_transparent_union_0(aLocal);
            Type_Attributes_transparent_union_0(bLocal);
        }
```

Supported in GNU mode only.

3.4.30   typeof

You can use typeof to generate a new item of the same type as an existing item. This is useful in macros where you have to create a temporary type, and you do not know the type that is to be used when the macro is written.

Supported in GNU mode only.

3.4.31   Unnamed fields

When embedding a structure or union within another structure or union, you do not have to name the internal structure. You can access the contents of the unnamed structure without using .name to reference it, for example:

```
struct {
    int a;
    union {
      int b;
      float c;
    };
    int d;
} Unnamed_Fields_0;
int Unnamed_Fields_1()
{
    return Unnamed_Fields_0.b;
}
```

Supported in GNU mode only. Unnamed fields are the same as anonymous unions and structures. See *Anonymous classes, structures and unions* on page 3-36 for details.

3.4.32   Variable attributes

This section describes the variable attributes that are supported:
*   *aligned* on page 3-84
*   *deprecated* on page 3-84
*   *packed* on page 3-85
*   *section* on page 3-85
*   *zero_init* on page 3-85
*   *transparent_union* on page 3-86

---

aligned

The aligned variable attribute specifies a minimum alignment for the variable or structure field, measured in bytes, for example:

```
int Variable_Attributes_aligned_0 __attribute__ ((aligned (16)));
/* aligned on 16 byte boundary */
short Variable_Attributes_aligned_1[3] __attribute__ ((aligned));
/* will align on 4 byte boundary for ARM */
```

Supported in GNU mode and ARM mode. Also, see *Type qualifiers* on page 3-26 for a description of __align(*n*).

deprecated

The deprecated variable attribute enables the declaration of a deprecated variable without any warnings or errors being issued by the compiler. However, any access to a deprecated variable creates a warning, but still compiles. The warning gives the location where the variable is used and the location where it is defined. This helps you to determine why a particular definition is deprecated.

```
extern int Variable_Attributes_deprecated_0 __attribute__ ((deprecated));
extern int Variable_Attributes_deprecated_1 __attribute__ ((deprecated));
void Variable_Attributes_deprecated_2()
{
    Variable_Attributes_deprecated_0=1;
    Variable_Attributes_deprecated_1=2;
}
```

Compiling this example, causes the following warnings to be generated:

```
Warning:  #1361-D: use of variable "Variable_Attributes_deprecated_0" (declared
at line 1) is deprecated
Warning:  #1361-D: use of variable "Variable_Attributes_deprecated_1" (declared
at line 2) is deprecated
```

Supported in GNU mode and ARM mode.

packed

The packed variable attribute specifies that a variable or structure field has the smallest possible alignment. That is, one byte for a variable, and one bit for a field, unless you specify a larger value with the aligned attribute.

```
struct
{
    char a;
    int b __attribute__ ((packed));
}Variable_Attributes_packed_0;
```

Supported in GNU mode and ARM mode.

section

Normally, the ARM compiler places the objects it generates in sections like data and bss. However, you might require additional data sections or you might want a variable to appear in a special section, for example, to map to special hardware. The section attribute specifies that a variable should be placed in a particular data section. If you use the section attribute, variables are placed in RW data sections unless you use the zero_init attribute. In this case, the variable is placed in a ZI section.

```
const int descriptor[3] __attribute__ ((section ("descr"))) = { 1,2,3 }; /* in
RW section */
long long rw[10] __attribute__ ((section ("RW")));   /* in RW section */
long long altstack[10] __attribute__ ((section ("STACK"), zero_init));   /* in
ZI section */
```

Supported in GNU mode and ARM mode.

zero_init

The section attribute specifies that a variable should be placed in a particular data section. The zero_init attribute specifies that a variable with no initializer is placed in a ZI data section. If an initializer is specified, an error is reported.

```
__attribute__((zero_init)) int x;                        /* in section ".bss"
*/__attribute__((section("mybss"), zero_init)) int y;  /* in section "mybss" */
```

Supported in GNU mode and ARM mode.

transparent_union

The `transparent_union` variable attribute, attached to a function parameter which is a union, means that the corresponding argument can have the type of any union member, but the argument is passed as if its type were that of the first union member.

You can also use this attribute on a `typedef` for a union data type. In this case it applies to all function parameters with that type.

——— **Note** ———

The C specification states that the value returned when a union is written as one type and read back with another is undefined. Therefore, a method of distinguishing which type a `transparent_union` was written in must also be passed as an argument.

———————————

```
typedef union
{
    int myint;
    float myfloat;
}transparent_union_t;
void Variable_Attributes_transparent_union_0(transparent_union_t aUnion
__attribute__ ((transparent_union)))
{
    static int aStatic;
    aStatic +=aUnion.myint;
}
void Variable_Attributes_transparent_union_1()
{
    int aLocal =0;
    float bLocal =0;
    Variable_Attributes_transparent_union_0(aLocal);
    Variable_Attributes_transparent_union_0(bLocal);
}
```

Supported in GNU mode only.

unused

Normally, the compiler warns if a variable is declared but is never referenced. This attribute informs the compiler that you expect a variable to be unused and to not issue a warning if it is not used.

```
void Variable_Attributes_unused_0()
{
    static int aStatic =0;
    int aUnused __attribute__ ((unused));
```

```
    int bUnused;
    aStatic++;
}
```

In this example, the compiler warns that `bUnused` is declared but never referenced, but does not warn about `aUnused`, for example:

```
Warning:  #177-D: variable "bUnused" was declared but never referenced
```

———— **Note** ————

The GNU compiler does not give any warning.

Supported in GNU mode and ARM mode.

weak

The declaration of a weak variable is permitted, and acts in a similar way to `__weak` (see *Function qualifiers* on page 3-9):

• in GNU mode:

```
extern int Variable_Attributes_weak_1 __attribute__ ((weak));
```

• The equivalent in ARM mode is:

```
__weak int Variable_Attributes_weak_compare;
```

Supported in GNU mode and ARM mode.

———— **Note** ————

The `extern` qualifier is required in GNU mode. In ARM mode the compiler assumes that if the variable is not `extern` then it is treated like any other non-weak variable.

3.4.33   Variadic macros

In C99 you can declare a macro to accept a variable number of arguments. The syntax for defining the macro is similar to that of a function, for example:

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
void Variadic_Macros_0()
{
    debug ("a test string is printed out along with %x %x %x\n", 12, 14, 20);
}
```

Supported in GNU mode and ARM mode.

3.4.34   Zero length arrays

You can use zero length arrays. These might be used for a header on a malloced data structure, for example:

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
typedef struct {
    int length;
    char letters[0];
    } linet;
linet* Zero_Length_0(char *arg)
{
    linet * linep = malloc (sizeof (linet) + strlen(arg) + 1 );
    strcpy (linep->letters, arg);
    return linep;
}
char * aString="this is a simple string";
int main() {
  printf("String: '%s'\n", Zero_Length_0(aString)->letters);
  return 0;
}
```

Supported in GNU mode only.

3.5   Predefined macros

Table 3-5 lists the macro names predefined by the ARM compiler for C and C++. Where the value field is empty, the symbol is only defined.

Compilation tools are restricted to the permitted endianness of the target MCU.

Table 3-5 Predefined macros

Name	Value	When defined
`__arm __arm__`	–	Always defined for the ARM compiler, even when you specify the `--thumb` option. See also `__ARMCC_VERSION`. ——— **Note** ——— These macros are also defined if you invoke the ARM compiler using the commands, `armcpp`, `tcc`, and `tcpp`.
`__ARMCC_VERSION`	*ver*	Always defined. It is a decimal number, and is guaranteed to increase between releases. The format is *PVtbbb* where: *P* is the major version (2 for RVCT v2.2)*V* is the minor version (2 for RVCT v2.2)*t* is the patch release (0 for RVCT v2.2, unpatched)*bbb* is the build (841 for example). The example given results in 220841. ——— **Note** ——— Use this to distinguish between RVCT and other tools that define `__arm__`.
`__APCS_ADSABI`	–	When you specify the `--apcs /adsabi` option.
`__APCS_INTERWORK`	–	When you specify the `--apcs /interwork` option or set the CPU architecture to ARMv5TE or later.
`__APCS_ROPI`	–	When you specify the `--apcs /ropi` option.
`__APCS_RWPI`	–	When you specify the `--apcs /rwpi` option.
`__APCS_FPIC`	–	When you specify the `--apcs /fpic` option.
`__APCS_SWST`	–	When you specify the `--apcs /swst` option.
`__ARRAY_OPERATORS`	–	In C++ compiler mode, to specify that array new and delete are enabled.

Table 3-5 Predefined macros

Name	Value	When defined
__BASE_FILE__	*name*	Always defined. Similar to __FILE__, but indicates the primary source file rather than the current one (that is, when the current file is an included file).
_BOOL	–	In C++ compiler mode, to specify that **bool** is a keyword.
c_plusplus	–	In C++ compiler mode, but not in strict mode.
__cplusplus	–	In C++ compiler mode.
__CC_ARM	1	Always set to 1 for the ARM compiler, even when you specify the --thumb option. ——— **Note** ——— This macro is also defined if you invoke the ARM compiler using armcpp, tcc, and tcpp.
__DATE__	*date*	When date of translation of source file is required.
__EDG__	–	Always defined.
__EDG_IMPLICIT_USING_STD	–	In C++ compiler mode when you specify the --implicit_using_std option.
__EDG_VERSION__	–	Always set to an integer value that represents the version number of the EDG front end. For example, version 3.0 is represented as 300.
__EXCEPTIONS	1	In C++ mode when you specify the --exceptions option.
__FEATURE_SIGNED_CHAR	–	When you specify the --signed_chars option (used by CHAR_MIN and CHAR_MAX).
__FILE__	*name*	The name of the current source file.
__FP_FAST	–	When you specify the --fpmode fast option.
__FP_FENV_EXCEPTIONS	–	When you specify the --fpmode ieee_full or --fpmode ieee_fixed options.
__FP_FENV_ROUNDING	–	When you specify the --fpmode ieee_full option.
__FP_IEEE	–	When you specify the --fpmode ieee_full, --fpmode ieee_fixed, or --fpmode ieee_no_fenv options.

Table 3-5 Predefined macros

Name	Value	When defined
`__FP_INEXACT_EXCEPTION`	–	When you specify the `--fpmode ieee_full` option.
`__FUNCTION__`	*name*	The name of the current function. This holds the function name, even if the function is inlined.
`__GNUC__`	*ver*	When you specify the `--gnu` option. It is an integer that shows the current major version of the GNU mode being used. For RVCT v2.2, the value is 3.
`__GNUC_MINOR__`	*ver*	When you specify the `--gnu` option. It is an integer that shows the current minor version of the GNU mode being used. For RVCT v2.2, the value is 0.
`__IMPLICIT_INCLUDE`	–	When you specify the `--implicit_include` option.
`__LINE__`	*num*	Always set. It is the source line number of the line of code containing this macro.
`__MODULE__`	*mod*	Contains the filename part of the value of `__FILE__`.
`__NO_INLINE__`	–	When you specify the `--no_inline` option in GNU mode.
`__OPTIMISE_LEVEL`	*num*	Always set to 2 by default, unless you change the optimization level using the `-0`*num* option.
`__OPTIMISE_SPACE`	–	When you specify the `-0space` option.
`__OPTIMISE_TIME`	–	When you specify the `-0time` option.
`__OPTIMIZE__`	–	When `-01` or `-02` (the default) is specified in GNU mode.
`__OPTIMIZE_SIZE__`	–	When `-0space` (the default) is specified in GNU mode.
`__PLACEMENT_DELETE`	–	In C++ compiler mode to specify that placement delete is enabled. This is only relevant when using exceptions.
`__PRETTY_FUNCTION__`	*name*	The unmangled name of the current function. This holds the unmangled function name, even if the function is inlined.

<div align="right">**Table 3-5 Predefined macros**</div>

Name	Value	When defined
`__RTTI`	–	In C++ compiler mode when RTTI is enabled.
`__sizeof_int`	4	For `sizeof(int)`, but available in preprocessor expressions.
`__sizeof_long`	4	For `sizeof(long)`, but available in preprocessor expressions.
`__sizeof_ptr`	4	For `sizeof(void *)`, but available in preprocessor expressions.
`__SOFTFP__`	–	If compiling to use the software floating-point calling standard and library. Set when you specify the `--fpu softvfp` option for ARM or Thumb, or when you specify `--fpu softvfp+vfpv2` for Thumb.
`__STDC__`	–	In all compiler modes.
`__STDC_VERSION__`	–	Standard version information.
`__STRICT_ANSI__`	–	When you specify the `--strict` option.
`__TARGET_ARCH_xx`	–	*xx* represents the target architecture and its value depends on the target architecture. For example, if you specify the compiler options `--cpu 5TE` or `--cpu ARM946E-S` then `__TARGET_ARCH_5TE` is defined, and no other symbol starting with `__TARGET_ARCH_` is defined.
`__TARGET_CPU_xx`	–	*xx* represents the target cpu. The value of *xx* is derived from the `--cpu` compiler option, or the default if none is specified. For example, if you specify the compiler option `--cpu ARM946E-S` then `__TARGET_CPU_ARM946E_S` is defined and no other symbol starting with `__TARGET_CPU_` is defined. ———— **Note** ———— The example shows that, if the processor name contains hyphen (-) characters (`ARM946E-S`), these are mapped to an underscore (`ARM946E-S`). If you specify the target architecture, then `__TARGET_CPU_generic` is defined.
`__TARGET_FEATURE_DOUBLEWORD`	–	If the target architecture supports the `PLD`, `LDRD`, `STRD`, `MCRR`, and `MRRC` instructions. That is, ARMv5T and later.

Table 3-5 Predefined macros

Name	Value	When defined
`__TARGET_FEATURE_DSPMUL`	–	If the DSP-enhanced multiplier is available, for example ARMv5TE.
`__TARGET_FEATURE_HALFWORD`	–	If the target architecture supports halfword and signed byte access instructions, for example ARMv4T and later.
`__TARGET_FEATURE_MULTIPLY`	–	If the target architecture supports the long multiply instructions `MULL` and `MULAL`.
`__TARGET_FEATURE_THUMB`	–	If the target architecture supports Thumb (ARMv4T or later).
`__TARGET_FPU_`*xx*	–	One of the following is set to indicate the FPU usage: • `__TARGET_FPU_NONE` • `__TARGET_FPU_SOFTVFP` See the description of the `--fpu` option in *Specifying the target processor or architecture* on page 2-39 for more information on FPU options.
`__thumb__thumb__`	–	When the compiler is in Thumb mode. That is, you have either specified the `--thumb` option on the command-line or `#pragma thumb` in your source code. ———— **Note** ———— • The compiler might generate some ARM code even if it is compiling for Thumb. • `__thumb` and `__thumb__` become defined or undefined when using `#pragma thumb` or `#pragma arm`, but do not change in cases where Thumb functions are generated as ARM for other reasons (for example, a function specified as `__irq`).
`__TIME__`	*time*	When time of translation of the source file is required.
`__VERSION__`	*ver*	When you specify the `--gnu` option. It is a string that shows the current version of the GNU mode being used. For RVCT v2.2, the value is `EDG gcc 3.0 mode`.
`_WCHAR_T`	–	In C++ compiler mode, to specify that **wchar_t** is a keyword.

Where you specify macro options, the compiler defines and undefines macros in the following order:

1. compiler predefined macros.

2. macros defined explicitly, using -D*name*.

3. macros explicitly undefined, using -U*name*.

See *Setting preprocessor options* on page 2-31 for details.

Chapter 4
Inline and Embedded Assemblers

This chapter describes the inline and embedded assemblers of the ARM® compiler, `armcc`. It contains the following sections:

- *Inline assembler* on page 4-2
- *Embedded assembler* on page 4-20
- *Legacy inline assembler that accesses sp, lr or pc* on page 4-29
- *Differences between inline and embedded assembly code* on page 4-31.

4.1 Inline assembler

The ARM compiler provides an inline assembler that enables you to write optimized assembly language routines, and access features of the target processor that are not available from C or C++.

──── **Note** ────

The inline assembler supports ARM assembly language only. It does not support Thumb assembly language. Use the embedded assembler for Thumb support.

────────────

The following sections are included:
- *Inline assembler syntax*
- *Restrictions on inline assembly operations* on page 4-8
- *Virtual registers* on page 4-11
- *Constants* on page 4-12
- *Instruction expansion* on page 4-12
- *Condition flags* on page 4-13
- *Operands* on page 4-13
- *Function calls and branches* on page 4-15
- *Labels* on page 4-17
- *Differences from previous versions of the ARM C/C++ compilers* on page 4-18.

For more information, see:

- *RealView Developer Kit v2.2 Assembler Guide* for detailed information on writing assembly language for the ARM processors.

4.1.1 Inline assembler syntax

The ARM compiler supports an extended inline assembler syntax, introduced by the `asm` keyword (C++), or the `__asm` keyword (C and C++). The syntax for these keywords is described in the following sections:
- *Inline assembly with the __asm keyword* on page 4-3
- *Inline assembly with the asm keyword* on page 4-3
- *Rules for using __asm and asm* on page 4-4
- *String copying using a loop* on page 4-5.

You can use an `asm` or `__asm` statement anywhere a statement is expected.

The inline assembler supports the majority of the ARM instruction set, including generic coprocessor instructions, and a subset of ARMv6 instructions. However, it does not support BX, BLX, and BXJ instructions. See *Restrictions on inline assembly operations* on page 4-8 for more details.

Inline assembly with the __asm keyword

The inline assembler is invoked with the assembler specifier, and is followed by a list of assembler instructions inside braces. You can specify inline assembler code using the following formats:

- On a single line, for example:

```
__asm("instruction[;instruction]"); // Must be a single string
__asm{instruction[;instruction]}
```

    You cannot include comments.

- On multiple lines, for example:

```
__asm{
    ...
    instruction
    ...
}
```

    You can use C or C++ comments anywhere in an inline assembly language block.

See *Rules for using __asm and asm* on page 4-4.

Inline assembly with the asm keyword

When compiling C++, the ARM compiler supports the **asm** syntax proposed in the ISO C++ Standard. You can specify inline assembler code using the following formats:

- On a single line, for example:

```
asm("instruction[;instruction]"); // Must be a single string
asm{instruction[;instruction]}
```

    You cannot include comments.

- On multiple lines, for example:

```
asm{
    ...
```

```
        instruction
        ...
    }
```

You can use C or C++ comments anywhere in an inline assembly language block.

Rules for using __asm and asm

Follow these rules when using the **__asm** and **asm** keywords:

- If you include multiple instructions on the same line, you must separate them with a semicolon. If you use double quotes, you must enclose all the instructions within a single set of double quotes.

- If an instruction requires more than one line, you must specify the line continuation with the backslash character \.

- For the multiple line format, you can use C or C++ comments anywhere in the inline assembly language block. However, you cannot embed comments in a line that contains multiple instructions.

- Comma is used as a separator in assembly language, so C expressions with the comma operator must be enclosed in parentheses to distinguish them:

  ```
  __asm {ADD x, y, (f(), z)}
  ```

- An **asm** statement must be inside a C++ function. An **asm** statement can be used anywhere a C++ statement is expected.

- Register names in the inline assembler are treated as C or C++ variables. They do not necessarily relate to the physical register of the same name (see *Virtual registers* on page 4-11). If you do not declare the register as a C or C++ variable, then the compiler warns you that it should be declared as a variable.

- Do not save and restore registers in inline assembler. The compiler does this for you. Also, the inline assembler does not provide direct access to the physical registers (see *Virtual registers* on page 4-11).

  If registers other than CPSR and SPSR are read without being written to, an error message is issued. For example:

  ```
  int f(int x)
  {
      __asm
      {
          STMFD sp!, {r0}    // save r0 - illegal: read before write
          ADD r0, x, 1
          EOR x, r0, x
  ```

```
            LDMFD sp!, {r0}    // restore r0 - not needed.
        }
        return x;
    }
```

The function must be written as:

```
int f(int x)
{
    int r0;
    __asm
    {
        ADD r0, x, 1
        EOR x, r0, x
    }
    return x;
}
```

Also, see *Restrictions on inline assembly operations* on page 4-8.

Examples

The following examples demonstrate some of the ways that you can use inline assembly language effectively:

- *String copying using a loop*
- *Enabling and disabling interrupts* on page 4-6
- *Dot product* on page 4-7.

String copying using a loop

Example 4-1 shows inline assembly code that copies a string value using a loop. This example is for illustration only, and is not an efficient byte copying routine. This code is also in the main examples directory, in ...\inline\strcopy.c.

Example 4-1 String copy with inline assembler

```
#include <stdio.h>
void my_strcpy(const char *src, char *dst)
{
    int ch;
    __asm
    {
    loop:
        LDRB    ch, [src], #1
        STRB    ch, [dst], #1
```

```
            CMP     ch, #0
            BNE     loop
    }
}
int main(void)
{
    const char *a = "Hello world!";
    char b[20];
    my_strcpy (a, b);
    printf("Original string: '%s'\n", a);
    printf("Copied   string: '%s'\n", b);
    return 0;
}
```

Enabling and disabling interrupts

Interrupts are enabled or disabled by reading the CPSR flags and updating bit 7.
Example 4-2 shows how this can be done by using small functions that can be inlined.

This code is also in the main examples directory, in ...\inline\irqs.c.

These functions work only in a privileged mode, because the control bits of the CPSR and
SPSR cannot be changed when in User mode.

Example 4-2 Interrupts

```
__inline void enable_IRQ(void)
{
    int tmp;
    __asm
    {
        MRS tmp, CPSR
        BIC tmp, tmp, #0x80
        MSR CPSR_c, tmp
    }
}
__inline void disable_IRQ(void)
{
    int tmp;
    __asm
    {
        MRS tmp, CPSR
        ORR tmp, tmp, #0x80
        MSR CPSR_c, tmp
    }
}
```

```
int main(void)
{
    disable_IRQ();
    enable_IRQ();
}
```

Dot product

Example 4-3 calculates the dot product of two integer arrays. It demonstrates how inline assembly language can interwork with C or C++ expressions and data types that are not directly supported by the inline assembler. The inline function `mlal()` is optimized to a single SMLAL instruction.

long long is the same as __int64.

This code is also in the main examples directory, in ...\inline\dotprod.c.

Example 4-3 Dot product

```
#include <stdio.h>
/* change word order if big-endian */
#define lo64(a) (((unsigned*) &a)[0])    /* low 32 bits of a long long */
#define hi64(a) (((int*) &a)[1])         /* high 32 bits of a long long */
__inline __int64 mlal(__int64 sum, int a, int b)
{
#if !defined(__thumb) && defined(__TARGET_FEATURE_MULTIPLY)
    __asm
    {
      SMLAL lo64(sum), hi64(sum), a, b
    }
#else
    sum += (__int64) a * (__int64) b;
#endif
    return sum;
}
__int64 dotprod(int *a, int *b, unsigned n)
{
    __int64 sum = 0;
    do
        sum = mlal(sum, *a++, *b++);
    while (--n != 0);
    return sum;
}
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int b[10] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
int main(void)
```

```
{
    printf("Dotproduct %lld (should be %d)\n", dotprod(a, b, 10), 220);
    return 0;
}
```

4.1.2   Restrictions on inline assembly operations

There are a number of restrictions on the operations that can be performed in inline assembly code. These restrictions provide a measure of safety, and ensure that the assumptions in compiled C and C++ code are not violated in the assembled assembly code.

The restrictions are described in the following sections:

*   *Miscellaneous restrictions*

*   *Registers* on page 4-9

*   *Processor modes* on page 4-9

*   *Thumb instruction set* on page 4-10

*   *Unsupported instructions* on page 4-10.

Miscellaneous restrictions

The inline assembler has the following restrictions:

*   The inline assembler is a high-level assembler, and the code it generates might not always be exactly what you write. Do not use it to generate more efficient code than the compiler generates. Use the ARM assembler armasm for this purpose.

*   Some low-level features that are available in the ARM assembler armasm, such as branching and writing to PC, are not supported.

*   Label expressions are not supported.

*   You cannot get the address of the current instruction using dot notation (.) or {PC}.

*   The & operator cannot be used to denote hexadecimal constants. Use the 0x prefix instead. For example:

    __asm {AND x, y, 0xF00}

*   The notation to specify the actual rotate of an 8-bit constant is not available in inline assembly language. This means that where an 8-bit shifted constant is used, the C flag must be regarded as corrupted if the NZCV flags are updated.

- You must not modify the stack. This is not necessary because the compiler stacks and restores any working registers as required automatically. It is not permitted to explicitly stack and restore work registers.

Registers

Registers, such as r0-r3, ip, lr, and the NZCV flags in the CPSR must be used with caution. If you use C or C++ expressions, these might be used as temporary registers and NZCV flags might be corrupted by the compiler when evaluating the expression. See *Virtual registers* on page 4-11.

The pc, lr, and sp registers cannot be explicitly read or modified using inline assembly code because there is no direct access to any physical registers. However, you can use the following intrinsics to access these registers:

__current_pc()

> To access the pc register. See *__current_pc() intrinsic* on page 3-16.

__current_sp()

> To access the sp register. See *__current_sp() intrinsic* on page 3-16.

__return_address()

> Although the lr register usually contains the return address, there is no guarantee that it does for inline assembly. For example, there are certain build options (such as software stack checking) or optimizations that might use lr for another purpose. Therefore, the __return_address() intrinsic always gives you a return address, which might be different from the lr register contents. See *__return_address() intrinsic* on page 3-16.

Also, see *Legacy inline assembler that accesses sp, lr or pc* on page 4-29.

Processor modes

You can change processor modes, alter the AAPCS registers fp, sl, and sb, or alter the state of coprocessors. However, the compiler is unaware of the change. If you change processor mode, you must not use C or C++ expressions until you change back to the original mode because the compiler corrupts the registers for the processor mode to which you have changed.

Similarly, if you change the state of a floating-point coprocessor by executing floating-point instructions, you must not use floating-point expressions until the original state has been restored.

Thumb instruction set

The inline assembler is not available when compiling C or C++ for Thumb state, and the inline assembler does not assemble Thumb instructions. If you attempt to compile inline assembly code for Thumb, the following error message is displayed:

```
Error:  #1113: Inline assembler not permitted when generating Thumb code
```

If you want to include inline assembly in code to be compiled for Thumb, enclose the functions containing inline assembler code between #pragma arm and #pragma thumb statements (see *Pragmas controlling code generation* on page 3-5). For example:

```
#pragma arm
int add(int i, int j) {
    int res;
    __asm {
        ADD   res, i, j   // add here
    }
    return res;
}
#pragma thumb
```

You must also compile your code using the --apcs /interwork compiler option (see *Interworking qualifiers* on page 2-65).

Unsupported instructions

The following instructions are not supported in the inline assembler:

- BKPT, BX, BXJ, and BLXinstructions.

  —————— **Note** ——————

  You can insert a BKPT instruction in C and C++ code by using the __breakpoint() intrinsic (see *__breakpoint() intrinsic* on page 3-15 for details).

  ——————————————

- the LDR Rn, =*expression* pseudo-instruction. Use MOV Rn, *expression* instead (this can generate a load from a literal pool).

- The ADR and ADRL pseudo-instructions.

4.1.3   Virtual registers

The inline assembler provides no direct access to the physical registers of an ARM processor. If an ARM register name is used as an operand in an inline assembler instruction it becomes a reference to a virtual register, with the same name, and not the physical ARM register.

The compiler allocates physical registers to each virtual register as appropriate during optimization and code-generation. However, the physical register used in the assembled code might be different to that specified in the instruction. You can explicitly define these virtual registers as normal C or C++ variables. If they are not defined then the compiler supplies implicit definitions for the virtual registers.

The compiler-defined virtual registers have function local scope, that is, within a single function, multiple **asm** statements or declarations that refer to the same virtual register name access the same virtual register.

No virtual registers are created for the sp (r13), lr (r14), and pc (r15) registers, and they cannot be read or directly modified in inline assembly code. See *Legacy inline assembler that accesses sp, lr or pc* on page 4-29 for details on how you can modify your source code.

There is no virtual *Processor Status Register* (PSR). Any references to the PSR are always to the physical PSR.

Existing inline assembler code that conforms to previously documented guidelines continues to perform the same function as before although the actual registers used in each instruction might be different to previous versions of the compiler.

The initial value in each virtual register is unpredictable. You must write to virtual registers before reading them. The compiler generates an error if you attempt to read a virtual register before writing to it. For example, if you attempt to read the virtual register associated with the variable r1:

```
Error:  #549: variable "r1" is used before its value is set
```

You must also explicitly declare the names of the variables in your C or C++ code. It is better to use C or C++ variables as instruction operands. A warning is given the first time a virtual or physical register name is used, and only once for each translation unit. For example, if you specify register r3, the following warning is displayed:

```
Warning #1267-D: Implicit physical register R3 should be defined as a variable
```

4.1.4   Constants

The constant expression specifier # is optional. If it is used, the expression following it must be a constant.

4.1.5   Instruction expansion

An ARM instruction in inline assembly code might be expanded into several instructions in the compiled object. The expansion depends on the instruction, the number of operands specified in the instruction, and the type and value of each operand.

Instructions using constants

The constant in instructions with a constant operand is not limited to the values permitted by the instruction. Instead, such an instruction is translated into a sequence of instructions with the same effect. For example:

```
ADD r0,r0,#1023
```

might be translated into:

```
ADD r0,r0,#1024
SUB r0,r0,#1
```

With the exception of coprocessor instructions, all ARM instructions with a constant operand support instruction expansion. In addition, the MUL instruction can be expanded into a sequence of adds and shifts when the third operand is a constant.

The effect of updating the CPSR by an expanded instruction is:

*   arithmetic instructions set the NZCV flags correctly.

*   logical instructions:
    —   set the NZ flags correctly
    —   do not change the V flag
    —   corrupt the C flag.

Load and store instructions

The LDM, STM, LDRD, and STRD instructions might be replaced by equivalent ARM instructions. The ARM compiler outputs a warning message informing you it might expand instructions, for example:

```
Warning:  #1287-D: LDM/STM instruction may be expanded
```

Inline assembly code must be written in a way that does not depend on the number of expected instructions or on the expected execution time for each specified instruction.

Instructions that normally place constraints on pairs of operand registers, such as LDRD and STRD, are replaced by a sequence of instructions with equivalent functionality and without the constraints. However, these might be recombined into LDRD and STRD instructions.

All LDM and STM instructions are expanded into a sequence of LDR and STR instructions with equivalent effect. However, the compiler might subsequently recombine the separate instructions into an LDM or STM during optimization.

4.1.6 Condition flags

An inline assembly instruction might explicitly or implicitly attempt to update the processor condition flags. Inline assembly instructions that involve only virtual register operands or simple expression operands (see *Operands*) have predictable behavior. The condition flags are set by the instruction if either an implicit or explicit update is specified. The condition flags are unchanged if no update is specified. If any of the instruction operands are not simple operands, then the condition flags might be corrupted unless the instruction updates them. In general, the compiler cannot easily diagnose potential corruption of the condition flags. However, for operands that require the construction and subsequent destruction of C++ temporaries the compiler gives a warning if the instruction attempts to update the condition flags. This is because the destruction might corrupt the condition flags.

4.1.7 Operands

Operands can be one of the following types:
- *Virtual registers*
- *Expression operands*
- *Register lists* on page 4-15
- *Intermediate operands* on page 4-15.

Virtual registers

Registers specified in inline assembly instructions always denote virtual registers and not the physical ARM integer registers. Virtual registers require no declaration, and the size of the virtual registers is the same as the physical registers. However, the physical register used in the assembled code might be different to that specified in the instruction. See *Virtual registers* on page 4-11 for more details.

Expression operands

Function arguments, C or C++ variables, and other C or C++ expressions can be specified as register operands in an inline assembly instruction.

The type of an expression used in place of an ARM integer register must be either an integral type (that is, `char`, `short`, `int` or `long`), excluding `long long`, or a pointer type. No sign extension is performed on `char` or `short` types. You must perform sign extension explicitly for these types. The compiler might add code to evaluate these expressions and allocate them to registers.

When an operand is used as a destination, the expression must be a modifiable lvalue if used as an operand where the register is modified. For example, a destination register or a base register with base-register update.

For an instruction containing more than one expression operand, the order that expression operands are evaluated is unspecified.

———— **Note** ————

No exceptions are thrown during expression evaluation.

An expression operand of a conditional instruction is only evaluated if the conditions for the instruction are met.

A C or C++ expression that is used as an inline assembler operand might result in the instruction being expanded into several instructions. This happens if the value of the expression does not meet the constraints set out for the instruction operands in the *ARM Architecture Reference Manual*.

If an expression used as an operand creates a temporary that requires destruction, then the destruction occurs after the inline assembly instruction is executed. This is analogous to the C++ rules for destruction of temporaries.

A simple expression operand is one of the following:
*   a variable value
*   the address of a variable
*   the dereferencing of a pointer variable
*   a compile-time constant.

Any expression containing one of the following is not a simple expression operand:
*   an implicit function call, such as for division, or explicit function call
*   the construction of a C++ temporary
*   an arithmetic or logical operation.

Register lists

A register list can contain a maximum of 16 operands. These operands can be virtual registers or expression register operands.

The order that virtual registers and expression operands are specified in a register list is significant. The register list operands are read or written in left-to-right order. The first operand uses the lowest address, and subsequent operands use addresses formed by incrementing the previous address by four. This new behavior is in contrast to the usual operation of the `LDM` or `STM` instructions where the lowest numbered physical register is always stored to the lowest memory address. This difference in behavior is a consequence of the virtualization of registers.

An expression operand or virtual register can appear more than once in a register list and is used each time it is specified.

The base register is updated, if specified. The update overwrites any value loaded into the base register during a memory load operation.

Operating on User mode registers when in a privileged mode, by specifying ∧ after a register list, is not supported by the inline assembler.

Intermediate operands

A C or C++ constant expression of an integral type might be used as an immediate value in an inline assembly instruction.

A constant expression that is used to specify an immediate shift must have a value that lies in the range defined in the *ARM Architecture Reference Manual*, as appropriate for the shift operation.

A constant expression that is used to specify an immediate offset for a memory or coprocessor data transfer instruction must have a value with suitable alignment.

4.1.8   Function calls and branches

The `BL` and `SWI` instructions of the inline assembler enable you to specify three optional lists following the normal instruction fields. These instructions have the following format:

```
SWI{cond} swi_num, {input_param_list}, {output_value_list}, {corrupt_reg_list}
BL{cond} function, {input_param_list}, {output_value_list}, {corrupt_reg_list}
```

The lists are described in the following sections:

- *No lists specified* on page 4-16

- *Input parameter list* on page 4-16
- *Output value list* on page 4-17
- *Corrupted register list* on page 4-17.

———— **Note** ————

- The BX, BLX, and BXJ instructions are not supported in the inline assembler.

- It is not possible to specify the lr, sp, or pc registers in any of the input, output, or corrupted register lists.

- The sp register must not be changed by any SWI instruction or function call.

No lists specified

If you do not specify any lists, then:
- r0-r3 are used as input parameters
- r0 is used for the output value
- r12 and r14 are corrupted.

Input parameter list

This list specifies the expressions or variables that are the input parameters to the function call or SWI instruction, and the physical registers that contain the expressions or variables. They are specified as assignments to physical registers or as physical register names. A single list can contain both types of input register specification.

The inline assembler ensures that the correct values are present in the specified physical registers before the BL or SWI instruction is entered. A physical register name that is specified without assignment ensures that the value in the virtual register of the same name is present in the physical register. This ensures backwards compatibility with existing inline assembler code.

For example, the instruction:

```
BL foo, {r0=expression1, r1=expression2, r2}
```

generates the following pseudo-code:

```
MOV (physical) r0, expression1
MOV (physical) r1, expression2
MOV (physical) r2, (virtual) r2
BL foo
```

Output value list

This list specifies the physical registers that contain the output values from the `BL` or `SWI` and where they must be stored. The output values are specified as assignments from physical registers to modifiable lvalue expressions or as single physical register names.

The inline assembler takes the values from the specified physical registers and assigns them into the specified expressions. A physical register name specified without assignment causes the virtual register of the same name to be updated with the value from the physical register.

For example, the instruction:

```
BL foo, { }, {result1=r0, r1}
```

generates the following pseudo-code:

```
BL foo
MOV result1, (physical) r0
MOV (virtual) r1, (physical) r1
```

Corrupted register list

This list specifies the physical registers that are corrupted by the called function. If the condition flags are modified by the called function then you must specify the `PSR` in the corrupted register list.

The `BL` and `SWI` instructions always corrupt `lr`.

If this list is omitted then for `BL` and `SWI`, the registers `r0-r3`, `ip`, `lr` and the `PSR` are corrupted.

The branch instruction, `B`, must only be used to jump to labels within a single C or C++ function.

4.1.9 Labels

Labels defined in inline assembly code can be used as targets for branches or C and C++ goto statements. Labels defined in C and C++ can be used as targets by branch instructions in inline assembly code, in the form:

```
BL{cond} label
```

4.1.10 Differences from previous versions of the ARM C/C++ compilers

There are significant differences between the inline assembler in the ARM compiler and the inline assembler in previous versions of the ARM C and C++ compilers. This section highlights the main areas of difference. It is expected that existing assembly code that conforms to the documented guidelines for inline assembler will continue to work as expected when compiled using the latest ARM compiler:

• see *ADS Developer Guide*, if you have ADS v1.2

ARMv6 instructions

Of all the ARMv6 instructions, the inline assembler supports the ARMv6 media instructions only.

Virtual registers

Inline assembly code for the compiler always specifies virtual registers. The compiler chooses the physical registers to be used for each instruction during code-generation, and enables the compiler to optimize fully the assembly code and surrounding C or C++ code.

The pc (r15), lr (r14), and sp (r13) registers cannot be accessed at all. An error message is generated when these registers are accessed.

The initial values of virtual registers are undefined. Therefore, you must write to virtual registers before reading them. The compiler warns you if code reads a virtual register before writing to it. The compiler also generates these warnings for legacy code that relies on particular values in physical registers at the beginning of inline assembly code, for example:

```
int add(int i, int j) {
    int res;
    __asm {
        ADD res, r0, r1   // relies on i passed in r0 and j passed in r1
    }
    return res;
}
```

This code generates the following warning and error messages:

```
Warning:  #1267-D: Implicit physical register R0 should be defined as a variable
Warning:  #1267-D: Implicit physical register R1 should be defined as a variable
Error:  #549: variable "R1" is used before its value is set
Error:  #549: variable "R0" is used before its value is set
```

The errors are generated because it reads virtual registers `r0` and `r1` before writing to them. The warnings are generated because `r0` and `r1` must be defined as C or C++ variables. The corrected code is:

```
int add(int i, int j) {
    int res;
    __asm {
        ADD res, i, j
    }
    return res;
}
```

Instruction expansion

The inline assembler in the compiler expands the instructions `LDM`, `STM`, `LDRD`, and `STRD` into a sequence of single-register memory operations that perform the equivalent functionality.

It is possible that the compiler optimizes the sequence of single-register memory operation instructions back into a multiple-register memory operation.

Register lists

The order of operands in a register list for an `LDM` or `STM` instruction is significant. They are used in the order given, that is left-to-right, and the first operand references the lowest generated memory address. This is in contrast to the behavior in previous compilers where the lowest numbered register always referenced the lowest memory address generated by the instruction.

This has changed because you can now use expression operands in register lists alongside virtual registers. The compiler gives a warning message if it encounters a register list that contains only virtual registers, and where the result of the new ordering is different to that from previous ARM C and C++ compilers.

Thumb instructions

The inline assembler in the compiler does not support the Thumb® instruction set. It does not assemble Thumb instructions, and cannot be used at all when compiling C or C++ for Thumb state, unless you use the `#pragma arm` and `#pragma thumb` pragmas (see *Thumb instruction set* on page 4-10).

4.2 Embedded assembler

The ARM compiler enables you to include assembly code out-of-line, in one or more C or C++ function definitions. Embedded assembler provides unrestricted, low-level access to the target processor, and enables you to use the C and C++ preprocessor directives and easy access to structure member offsets.

The following sections are included:
- *Embedded assembler syntax*
- *Restrictions on embedded assembly* on page 4-22
- *Differences between expressions in embedded assembly and C or C++* on page 4-23
- *Generation of embedded assembly functions* on page 4-23
- *The __cpp keyword* on page 4-25
- *Manual overload resolution* on page 4-25
- *Keywords for related base classes* on page 4-25
- *Keywords for member function classes* on page 4-26
- *Calling non-static member functions* on page 4-27
- *Differences from previous versions of the ARM C/C++ compilers* on page 4-28.

For more information, see *RealView Developer Kit v2.2 Assembler Guide* for detailed information on writing assembly language for the ARM processors.

4.2.1 Embedded assembler syntax

An embedded assembly function definition is marked by the `__asm` (C and C++) or `asm` (C++) function qualifiers, and can be used on:
- member functions
- non-member functions
- template functions
- template class member functions.

Functions declared with `__asm` or `asm` can have arguments, and return a type. They are called from C and C++ in the same way as normal C and C++ functions. The syntax of an embedded assembly function is:

```
__asm return-type function-name(parameter-list)
{
    // ARM/Thumb assembler code
    instruction[;instruction]
```

```
    ...
    instruction
}
```

The initial state of the embedded assembler (ARM or Thumb) is determined by the initial state of the compiler, as specified on the command line. That is:

- if the compiler starts in ARM state, the embedded assembler uses `--arm`
- if the compiler starts in Thumb state, the embedded assembler uses `--thumb`
- the embedded assembler state is not affected by `#pragma arm` and `#pragma thumb` pragmas.

You can change the state of the embedded assembler by using explicit `ARM`, `THUMB`, or `CODE16` directives in the embedded assembler function. The directives have effect until the next such directive is encountered. That is, a directive will set ARM or Thumb state for all subsequent `__asm` functions until it is over-ridden.

———— **Note** ————

Argument names are permitted in the parameter list, but they cannot be used in the body of the embedded assembly function. For example, the following function uses integer `i` in the body of the function, but this is not valid in assembly:

```
__asm int f(int i) {
    ADD i, i, #1 // error
}
```

You can use, for example, `r0` instead of `i`.

———————————————

Embedded assembler example

Example 4-4 is equivalent to Example 4-1 on page 4-5, but modified to show how to use the string copy routine as an embedded assembler routine.

Example 4-4 String copy with embedded assembler

```
#include <stdio.h>
__asm void my_strcpy(const char *src, const char *dst) {
loop
     LDRB  r3, [r0], #1
     STRB  r3, [r1], #1
     CMP   r3, #0
     BNE   loop
     MOV   pc, lr
```

```
}
void main()
{
    const char *a = "Hello world!";
    char b[20];
    my_strcpy (a, b);
    printf("Original string: '%s'\n", a);
    printf("Copied   string: '%s'\n", b);
}
```

4.2.2   Restrictions on embedded assembly

The following restrictions apply to embedded assembly functions:

- After preprocessing, __asm functions can only contain assembly code, with the exception of the following identifiers (see *Keywords for related base classes* on page 4-25 and *Keywords for member function classes* on page 4-26):

    ```
    __cpp(expr)
    __offsetof_base(D, B)
    __mcall_is_virtual(D, f)
    __mcall_is_in_vbase(D, f)
    __mcall_offsetof_base(D, f)
    __mcall_this_offset(D, f)
    __vcall_offsetof_vfunc(D, f)
    ```

- No return instructions are generated by the compiler for an __asm function. If you want to return from an __asm function, then you must include the return instructions, in assembly code, in the body of the function.

    ———— **Note** ————

    This makes it possible to fall through to the next function, because the embedded assembler guarantees to emit the __asm functions in the order you have defined them. However, inlined and template functions behave differently (see *Generation of embedded assembly functions* on page 4-23).

- __asm functions do not change the AAPCS rules that apply. This means that all calls between an __asm function and a normal C or C++ function must adhere to the AAPCS, even though there are no restrictions on the assembly code that an __asm function can use (for example, change state).

4.2.3 Differences between expressions in embedded assembly and C or C++

Be aware of the following differences between embedded assembly and C or C++:

- Assembler expressions are always unsigned. The same expression might have different values between assembler and C or C++. For example:

```
MOV r0, #(-33554432 / 2)        // result is 0x7f000000
MOV r0, #__cpp(-33554432 / 2)   // result is 0xff000000
```

- Assembler numbers with leading zeros are still decimal. For example:

```
MOV r0, #0700            // decimal 700
MOV r0, #__cpp(0700)     // octal 0700 == decimal 448
```

- Assembler operator precedence differs from C and C++. For example:

```
MOV r0, #(0x23 :AND: 0xf + 1)    // ((0x23 & 0xf) + 1) => 4
MOV r0, #__cpp(0x23 & 0xf + 1)   // (0x23 & (0xf + 1)) => 0
```

- Assembler strings are not null-terminated:

```
DCB "no trailing null"                  // 16 bytes
DCB __cpp("I have a trailing null!!")   // 25 bytes
```

——— **Note** ———

The assembler rules apply outside of __cpp, and the C or C++ rules apply inside __cpp. See *The __cpp keyword* on page 4-25 for more details on __cpp keyword.

4.2.4 Generation of embedded assembly functions

The bodies of all the __asm functions in a translation unit are assembled as if they are concatenated into a single file that is then passed to the ARM assembler. The order of __asm functions in the assembly file that is passed to the assembler is guaranteed to be the same order as in the source file, except for functions that are generated using a template instantiation.

——— **Note** ———

This means that it is possible for control to pass from one __asm function to another by falling off the end of the first function into the next __asm function in the file, if the return instruction is omitted.

When you invoke `armcc`, the object file produced by the assembler is combined with the object file of the compiler by a partial link that produces a single object file.

The compiler generates an `AREA` directive for each `__asm` function. For example, the following `__asm` function:

```
#include <cstddef>
struct X { int x,y; void addto_y(int); };
__asm void X::addto_y(int) {
    LDR     r2,[r0, #__cpp(offsetof(X, y))]
    ADD     r1,r2,r1
    STR     r1,[r0, #__cpp(offsetof(X, y))]
    BX      lr
}
```

For this function, the compiler generates:

```
    AREA ||.emb_text||, CODE, READONLY
    EXPORT |_ZN1X7addto_yEi|
#line num "file"
|_ZN1X7addto_yEi| PROC
    LDR r2,[r0, #4]
    ADD r1,r2,r1
    STR r1,[r0, #4]
    BX lr
    ENDP
    END
```

The use of `offsetof` must be inside the `__cpp()` because it is the normal `offsetof` macro from the `cstddef` header file.

Ordinary `__asm` functions are put in an ELF section with the name `.emb_text`. That is, embedded assembly functions are never inlined. However, implicitly instantiated template functions and out-of-line copies of inline functions are placed in an area with a name that is derived from the name of the function, and an extra attribute that marks them as common. This ensures that the special semantics of these kinds of functions is maintained.

——— **Note** ———

Due to the special naming of the area for out-of-line copies of inline functions and template functions, these functions are not in the order of definition, but in an arbitrary order. Therefore, you cannot assume that a code execution falls out of an inline or template function and into another `__asm` function.

———————————

4.2.5   The __cpp keyword

You can use the **__cpp** keyword to access C or C++ compile-time constant expressions, including the addresses of data or functions with external linkage, from the assembly code. The expression inside the __cpp must be a constant expression suitable for use as a C++ static initialization (see section *3.6.2 Initialization of non-local objects* and section *5.19 Constant expressions* in ISO/IEC 14882:1998).

The use of __cpp(*expr*) is replaced by a constant that can be used by the assembler. For example:

```
LDR r0, =__cpp(&some_variable)
LDR r1, =__cpp(some_function)
BL  __cpp(some_function)
MOV r0, #__cpp(some_constant_expr)
```

Names in the __cpp expression are looked up in the C++ context of the __asm function. Any names in the result of a __cpp expression are mangled as required and automatically have IMPORT statements generated for them.

4.2.6   Manual overload resolution

You can use C++ casts to do overload resolution for non-virtual function calls. For example:

```
void g(int);
void g(long);
struct T {
    int mf(int);
    int mf(int,int);
};
__asm void  f(T*, int, int) {
    BL __cpp(static_cast<int (T::*)(int, int)>(&T::mf)) // calls T::mf(int, int)
    BL __cpp(static_cast<void (*)(int)>(g)) // calls g(int)
    BX lr
}
```

4.2.7   Keywords for related base classes

The following keyword enables you to determine the offset from the beginning of an object to a related base class within it:

__offsetof_base(D, B)

> B must be a non-virtual base class of D.

Returns the offset from the beginning of a D object to the start of the B base subobject within it. The result might be zero. It is the offset (in bytes) that must be added to a D* p to implement the equivalent of static_cast<B*>(p). For example:

```
__asm B* my_static_base_cast(D* /*p*/) {
    if __offsetof_base(D, B) <> 0 // optimise zero offset case
        ADD r0, r0, #__offsetof_base(D, B)
    endif
    BX lr
}
```

These keywords are converted into integer or logical constants in the assembler source. You can only use them in __asm functions, but not in a __cpp expression.

4.2.8   Keywords for member function classes

The following keywords facilitate the calling of virtual and non-virtual member functions from an __asm function. The keywords beginning with __mcall can be used for both virtual and non-virtual functions. The keywords beginning with __vcall can be used only with virtual functions. The keywords do not particularly help in calling static member functions.

For examples of how to use these keywords, see *Calling non-static member functions* on page 4-27.

__mcall_is_virtual(D, f)

> Results in {TRUE} if f is a virtual member function found in D, or a base class of D, otherwise {FALSE}. If it returns {TRUE} the call can be done using virtual dispatch, otherwise the call must be done directly.

__mcall_is_in_vbase(D, f)

> Results in {TRUE} if f is non-static member function found in a virtual base class of D, otherwise {FALSE}. If it returns {TRUE} the this adjustment must be done using __mcall_offsetof_vbase(D, f), otherwise it must be done with __mcall_this_offset(D, f).

__mcall_offsetof_vbase(D, f)

> Where D class type and f is a non-static member function defined a virtual base class of D, in other words __mcall_is_in_vbase(D,f) returns true.
>
> This returns at which negative offset in the vtable of the vtable slot that holds the base offset (from the beginning of a D object to the start of the base in which f is defined).

This is the `this` adjustment necessary when making a call to `f` with a pointer to a `D`.

──── **Note** ────

The offset returns a positive number that then has to be subtracted from the vtable pointer.

────────────────

`__mcall_this_offset(D, f)`

Where `D` class type and `f` is a non-static member function defined in `D` or a non-virtual base class of `D`.

This returns the offset from the beginning of a `D` object to the start of the base in which `f` is defined. This is the this adjustment necessary when making a call to `f` with a pointer to a `D`. It is either zero if `f` is found in `D` or the same as `__offsetof_base(D,B)`, where `B` a non-virtual base class of `D` that contains `f`.

If `__mcall_this_offset(D,f)` is used when `f` is found in a virtual base class of `D` it returns an arbitrary value designed to cause an assembly error if used. This is so that such invalid uses of `__mcall_this_offset` can occur in sections of assembly code that are to be skipped.

`__vcall_offsetof_vfunc(D, f)`

Where `D` is a class and `f` is a virtual function defined in `D`, or a base class of `D`.

This returns an offset into the virtual function table at which the offset from the virtual function table to the virtual function can be found.

If `__vcall_offsetof_vfunc(D, f)` is used when `f` is not a virtual member function it returns an arbitrary value designed to cause an assembly error if used.

4.2.9 Calling non-static member functions

You can use these keywords to call virtual and non-virtual functions from `__asm` functions. There is no `__mcall_is_static` to detect static member functions because static member functions have different parameters (that is, no **this**), and therefore call sites are likely to already be specific to calling a static member function.

Calling a non-virtual member function

For example, the following code can be used to call a virtual function in either a virtual or non-virtual base:

```
                        // rp contains a D* and we wish to do the equivalent of rp->f() where f is a
                        // non-virtual function
                        // all arguments other than the this pointer are already setup
                        // assumes f does not return a struct
                        if __mcall_is_in_vbase(D, f)
                            LDR r12, [rp]                               // fetch vtable pointer
                            ldr r0, [r12, #-__mcall_offsetof_vbase(D, f)] // fetch the base offset
                            ADD r0, r0, rp                              // do this adjustment
                        else
                            MOV r0, rp                                  // set up this pointer for D*
                            LDR r12, [rp]                               // fetch vtable pointer
                            ADD r0, r0, #__mcall_this_offset(D, f)      // do this adjustment
                        endif
                            MOV lr, pc                                  // prepare lr
                            LDR pc, [r12, #__vcall_offsetof_vfunc(D, f)] // calls rp->f()
```

Calling a virtual member function

For example, the following code can be used to call a virtual function in either a virtual or non-virtual base:

```
                        // rp contains a D* and we wish to do the equivalent of rp->f() where f is a
                        // virtual function
                        // all arguments other than the this pointer are already setup
                        // assumes f does not return a struct
                        if __mcall_is_in_vbase(D, f)    LDR r12, [rp]                               //
                        fetch vtable pointer    ldr r0, [r12, #-__mcall_offsetof_vbase(D, f)] // fetch
                        the base offset    ADD r0, r0, rp                              // do this
                        adjustmentelse    MOV r0, rp                                  // set up this
                        pointer for D*    LDR r12, [rp]                               // fetch vtable
                        pointer        ADD r0, r0, #__mcall_this_offset(D, f)      // do this
                        adjustmentendif    MOV lr, pc                                 // prepare lr
                        LDR pc, [r12, #__vcall_offsetof_vfunc(D, f)]  // calls rp->f()
```

4.2.10 Differences from previous versions of the ARM C/C++ compilers

This section describes the main areas of difference between the embedded assembler in the ARM compiler and the embedded assembler in previous versions of the ARM C and C++ compilers. It is expected that existing assembly code that conforms to the documented guidelines for embedded assembler will continue to work as expected when compiled using the latest ARM compiler.

ARMv6 instructions

The embedded assembler supports all ARMv6 instructions.

4.3 Legacy inline assembler that accesses sp, lr or pc

The compilers in RVCT v1.2 and earlier enabled accesses to sp (r13), lr (r14), and pc (r15) from inline assembly code (see *Inline assembler* on page 4-2). Example 4-5 shows how legacy inline assembly code might use lr.

Example 4-5 Legacy inline assembly code using lr

```
void func()
{
    int var;
    __asm
    {
        mov  var, lr  /* get the return address of func() */
    }
}
```

There is no guarantee that lr contains the return address of a function if your legacy code uses it in inline assembly. For example, there are certain build options (such as software stack checking) or optimizations that might use lr for another purpose.

The compiler in RVCT v2.1 and later reports the following error if sp, lr or pc is used in this way, for example:

```
Error:  #20: identifier "lr" is undefined
```

If you have to access these registers from within a C or C++ source file, you can:

* use embedded assembly (see *Embedded assembler* on page 4-20).

* use the following intrinsics in inline assembly:

    __current_pc()

    > To access the pc register. See *__current_pc() intrinsic* on page 3-16.

    __current_sp()

    > To access the sp register. See *__current_sp() intrinsic* on page 3-16.

    __return_address()

    > To access the lr register. See *__return_address() intrinsic* on page 3-16.

See *Accessing sp (r13), lr (r14), and pc (r15) in legacy code* on page 4-30 for solutions that enable you to access these registers.

4.3.1   Accessing sp (r13), lr (r14), and pc (r15) in legacy code

The following solutions enable you to access the sp, lr, and pc registers correctly in your source code:

Solution 1   Use the compiler intrinsics in inline assembly, for example:

```
void printReg()
{
    unsigned int spReg, lrReg, pcReg;
    __asm {
        MOV spReg, __current_sp()
        MOV pcReg, __current_pc()
        MOV lrReg, __return_address()
    }
    printf("SP = 0x%X\n",spReg);
    printf("PC = 0x%X\n",pcReg);
    printf("LR = 0x%X\n",lrReg);
}
```

Solution 2   Use embedded assembly to access physical ARM registers from within a C or C++ source file (see *Embedded assembler* on page 4-20), for example:

```
__asm void func()
{
    MOV r0, lr
    ...
    BX lr
}
```

This enables the return address of a function to be captured and displayed, for example, for debugging purposes, to show the call tree.

———— **Note** ————

Be aware that the compiler might also inline a function into its caller function. If a function is inlined, then the return address is the return address of the function that calls the inlined function. Also, a function might be tail called. See *__return_address() intrinsic* on page 3-16 for more details.

4.4 Differences between inline and embedded assembly code

There are differences between the way inline and embedded assembly is compiled:

- Inline assembly code uses a high-level of processor abstraction, and is integrated with the C and C++ code during code generation. Therefore, the compiler optimizes the C and C++ code, and the assembly code together.

- Unlike inline assembly code, embedded assembly code is assembled separately from the C and C++ code to produce a compiled object that is then combined with the object from the compilation of the C or C++ source.

- Inline assembly code can be inlined by the compiler, but embedded assembly code cannot be inlined, either implicitly or explicitly.

Table 4-1 summarizes the main differences between inline assembler and embedded assembler.

Table 4-1 Differences between inline and embedded assembler

Feature	Embedded assembler	Inline assembler
Instruction set	ARM and Thumb.	ARM only.
ARM assembler directives	All supported.	None supported.
ARMv6 instructions	All supported.	Supports only the media instructions.
C/C++ expressions	Constant expressions only.	Full C/C++ expressions.
Optimization of assembly code	No optimization.	Full optimization.
Inlining	Never.	Possible.
Register access	Specified physical registers are used. You can also use `PC`, `LR` and `SP`.	Uses virtual registers (see *Virtual registers* on page 4-11). Using `sp` (`r13`), `lr` (`r14`), and `pc` (`r15`) gives an error. See *Legacy inline assembler that accesses sp, lr or pc* on page 4-29.
Return instructions	You must add them in your code.	Generated automatically. (The `BX`, `BXJ`, and `BLX` instructions are not supported.)
`BKPT` instruction	Supported directly.	Not supported.

—— **Note** ——

See also *Differences between expressions in embedded assembly and C or C++* on page 4-23.

Chapter 5
The C and C++ Libraries

This chapter describes the ARM® C and C++ libraries. The libraries support programs written in C or C++. This chapter contains the following sections:

- *About the runtime libraries* on page 5-2
- *Writing reentrant and thread-safe code* on page 5-6
- *Building an application with the C library* on page 5-20
- *Building an application without the C library* on page 5-27
- *Tailoring the C library to a new execution environment* on page 5-35
- *Tailoring static data access* on page 5-45
- *Tailoring locale and CTYPE* on page 5-46
- *Tailoring error signaling, error handling, and program exit* on page 5-71
- *Tailoring storage management* on page 5-77
- *Tailoring the runtime memory model* on page 5-87
- *Tailoring the input/output functions* on page 5-95
- *Tailoring other C library functions* on page 5-106
- *Selecting real-time division* on page 5-111
- *ISO implementation definition* on page 5-112
- *C library extensions* on page 5-121
- *Library naming conventions* on page 5-128.

Copyright © 2005 ARM Limited. All rights reserved.

5.1 About the runtime libraries

The following runtime libraries are provided to support compiled C and C++:

C   The C libraries consist of:

- 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 when compiling C and C++.

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 2.02.03
- helper functions when compiling C++
- additional C++ functions not supported by the Rogue Wave library.

——— **Note** ———

Support for C++ may be obtained as an upgrade on certain versions of RVDK, by means of an appropriate FLEX*lm* license.

For a detailed description of how the libraries comply with the ISO standard, see *ISO implementation definition* on page 5-112.

This section includes:

- *Features of the C and C++ libraries* on page 5-3
- *Namespaces* on page 5-3
- *Build options and library variants* on page 5-4
- *Library directory structure* on page 5-4
- *Using the VFP support libraries* on page 5-5
- *Thumb C libraries* on page 5-5.

RVDK uses an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The file format for each version of RVDK is restricted to the proprietary ATPE format for the permitted device. This is referred to as *ATPE_Custom*.

5.1.1 Features of the C and C++ libraries

In *RealView® Developer Kit* (RVDK), the C libraries use the standard ARM semihosted environment to provide facilities such as file input/output. This environment is supported by RVI-ME. See the description of semihosting in Chapter 7 *Semihosting* for more information on the debug environment.

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.

You can also tailor many of the target-independent functions to your own application-specific requirements, for example:

- the `malloc` family
- the `ctype` family
- all the locale-specific functions.

Many of the C library functions are independent of any other function and contain no target dependencies. You can easily exploit these functions from assembly language.

5.1.2 Namespaces

All C++ standard library names are defined in the namespace `std`, including the C library names when you include them using the following C++ syntax:

```
#include <cstdlib> // instead of stdlib.h
```

This means that you must qualify all the library names by using one of the following methods:

- specify the standard namespace, for example:

```
std::printf("example\n");
```

- use the C++ keyword **using** to import a name to the global namespace:

```
using namespace std;
printf("example\n");
```

- use the compiler option `--using_std`.

———— **Note** ————

`errno` is a macro, so it is not necessary to qualify it with a namespace.

5.1.3 Build options and library variants

When you build your application, you must make certain choices. For example:

Byte order Big-endian or little-endian. (On this product, only little-endian is supported.)

Floating-point support

VFP, software, or none.

Stack limit Checked or unchecked.

Position-independence

Data can be read/write position-independent or position-dependent.

Code can be read-only position-independent or position-dependent.

When you link your assembly language, C, or C++ code, the linker selects appropriate C and C++ library variants compatible with the build options you specified. There is a variant of the ISO C library for each combination of major build options. Build options are described in more detail in:

- the chapter on creating and using libraries in *RealView Developer Kit v2.2 Linker and Utilities Guide*

- *Using AAPCS options* on page 2-64 for the compiler

- *RealView Developer Kit v2.2 Assembler Guide* for the assembler.

5.1.4 Library directory structure

The libraries are installed in two subdirectories within RVCT library directory `...\lib`:

`armlib` This subdirectory contains the variants of the ARM C library, the floating-point arithmetic library (fplib), and the math library (mathlib). The accompanying header files are in `...\include`.

`cpplib` This subdirectory contains the variants of the Rogue Wave C++ library (cpp_*) and supporting ARM C++ functions (cpprt_*), referred to collectively as the ARM C++ Libraries. The accompanying header files are installed in `...\include`.

The environment variable `RVCT22LIB` must be set to point to the `lib` directory, or if this variable is not set, `ARMLIB`. Alternatively use the `--libpath` argument to the linker to identify the directory holding the library subdirectories. You do not have to identify the `armlib` and `cpplib` directories separately. The linker finds them for you from the location of `lib`.

———— **Note** ————

- The ARM C libraries are supplied in binary form only.

- 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, and include it when you link the application. Your version of the function is used instead of the standard library version.

- Normally, only a few functions in the ISO C library require re-implementation 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 license fee.

5.1.5 Using the VFP support libraries

You do not use the VFP support libraries (`vfpsupport`) by linking them directly to your application. You use them by invoking them from the undefined instruction trap in some platform-specific way. For example, your operating system might link to them, or some other kind of board setup software.

Example code is provided in the main examples directory, in `...\vfpsupport`. This example shows you how to set up the undefined instruction handler to invoke the VFP support code.

5.1.6 Thumb C libraries

The linker automatically links in the Thumb® C library if it detects that one or more of the objects to be linked have been built for:

- Thumb, either using the `--thumb` option or `#pragma thumb`
- interworking, using the `--apcs /interwork` option on architecture ARMv4T or above.

Despite its name, the Thumb C library does not contain exclusively Thumb code, but uses ARM instructions for critical functions, such as `memcpy`, `memset`, and `memclr`, for good performance. The bulk of the Thumb C library, however, is coded in Thumb for the best code density.

5.2 Writing reentrant and thread-safe code

The ARM C libraries support multithreading, for example where you are using a *Real Time Operating System* (RTOS). When describing multithreading in the rest of this section, we use the following definitions:

Threads   Used to mean multiple streams of execution sharing global data between them.

Process   Used to mean a collection of all the threads that share a particular set of global data.

If there are multiple processes on a machine, they can be entirely separate and do not share any data (except under unusual circumstances). Similarly, each process might be a single-threaded process or might be divided into multiple threads.

Where you have single-threaded processes, there is only one flow of control. In multithreaded applications, however, several flows of control might try to access the same functions, and the same resources, concurrently. To protect the integrity of resources, any code you write for multithreaded applications must be *reentrant* and *thread-safe*.

This section contains information to help you write multithreaded applications:
* *Introduction to reentrancy and thread-safety*
* *Use of static data in the C libraries* on page 5-7
* *The __user_libspace static data area* on page 5-8
* *Managing locks in multithreaded applications* on page 5-9
* *Using the ARM C libraries with a multithreaded application* on page 5-10
* *Thread-safety in the ARM C libraries* on page 5-12
* *Thread-safety in the ARM C++ libraries* on page 5-18.

5.2.1 Introduction to reentrancy and thread-safety

Reentrancy and thread-safety are both related to the way functions handle resources. However, they are different:

* A reentrant function does not hold static data over successive calls, nor does it return a pointer to static data. For this type of function, the caller provides all the data that the function requires, such as pointers to any workspace. This means that multiple concurrent invocations of the function do not interfere with each other.

    ——— **Note** ———
    A reentrant function must not call non-reentrant functions.

- A thread-safe function protects shared resources from concurrent access using *locks*. Thread-safety concerns only how a function is implemented and not its external interface. In C, local variables are dynamically allocated on the stack. Therefore, any function that does not use static data, or other shared resources, is usually thread-safe.

5.2.2   Use of static data in the C libraries

Static data refers to persistent read/write data that is not stored on the stack or the heap. This persistent data can be external or internal in scope, and is:

- at a fixed address, when compiled with `--apcs /norwpi`
- at a fixed address relative to the static base (sb) register (r9), when compiled with `--apcs /rwpi`.

Libraries that use static data might be reentrant, but this depends on their use of the `__user_libspace` static data area, and on the build options you choose:

- When compiled with `--apcs /norwpi`, read/write static data is addressed in a position-dependent fashion. This is the compiler default. Code from these variants is single-threaded because it uses read/write static data.

  For example, library `c_a__un` has position-dependent data (see *Library naming conventions* on page 5-128).

- When compiled with `--apcs /rwpi`, read/write static data is addressed in a position-independent fashion using offsets from the static base register sb. Code from these variants is reentrant, and can be multiply threaded if each thread uses a different static base value.

  For example, library `c_a__ue` has position-independent data (see *Library naming conventions* on page 5-128).

The following describes how the C libraries use static data:

- The default floating-point arithmetic libraries `fz_*` and `fj_*` do not use static data and are always reentrant. However, the `f_*` and `g_*` libraries do use static data.

- All statically-initialized data in the C libraries is read-only.

- All writable static data is zero initialized.

- Most C library functions use no writable static data and are reentrant whether built with default build options (`--apcs /norwpi`) or reentrant build options (`--apcs /rwpi`).

- Some functions have static data implicit in their definitions. You must not use these in a reentrant application unless you build it with `--apcs /rwpi` and the caller uses different values in `sb`.

See *Position independence qualifiers* on page 2-65 for details on the `--apcs` build options described here.

——— **Note** ———

Exactly which functions use static data in their definitions might change in future releases.

5.2.3 The __user_libspace static data area

The `__user_libspace` static data area holds the static data for the C libraries. This is a block of 96 bytes in the ZI segment, supplied by the C library. It is also used as a temporary stack during C library initialization.

The default ARM C libraries use the `__user_libspace` area to hold:

- `errno`, used by any function that is capable of setting `errno`. By default, `__rt_errno_addr()` returns a pointer to `errno`.

- The FP status word for software floating-point (exception flags, rounding mode). It is unused in hardware floating-point. By default, `__rt_fp_status_addr()` returns a pointer to the FP status word.

- A pointer to the base of the heap (that is, the `__Heap_Descriptor`), used by all the `malloc`-related functions.

- The `alloca` state, used by `alloca()` and its helper functions.

- The current locale settings, used by functions such as `setlocale()`, but also used by all other library functions which depend on them. For example, the `ctype.h` functions have to access the `LC_CTYPE` setting.

The C++ libraries use the `__user_libspace` area to hold:

- The `new_handler` field and `ddtor_pointer`:
    - the `new_handler` field is used to keep track of the value passed to `std::set_new_handler()`
    - the `ddtor_pointer` is used by `__cxa_atexit()` and `__aeabi_atexit()`.

- C++ exception handling information for functions such as `std::set_terminate()` and `std::set_unexpected()`.

See the *C++ ABI for the ARM Architecture* and *Exception Handling ABI for the ARM Architecture* specifications for more information on `__aeabi_atexit()`, `std::set_terminate()` and `std::set_unexpected()`.

———— **Note** ————

How the C and C++ library use the `__user_libspace` area might change in future releases.

Addressing __user_libspace

Two wrapper functions are provided to return a subsection of the `__user_libspace` static data area:

__user_perproc_libspace()

> This returns a pointer to 96 bytes 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 96 bytes used to store data that is local to a particular thread. This means that `__user_perthread_libspace()` returns a different address depending on the thread it is called from.

Re-implementing __user_libspace

The `__user_libspace()` function does not normally have to be redefined. However, if you are writing an operating system or a process switcher, you must re-implement this function (see *Tailoring static data access* on page 5-45 for more details).

Where you are porting single-threaded processes to RVCT v2.2 that re-implement this function, you can continue to do this without having to change your code. The change in behavior is, however, important if you are using RVCT to write multithreaded applications.

5.2.4 Managing locks in multithreaded applications

A thread-safe function protects shared resources from concurrent access using locks. In RVCT v2.2, and later, there are functions that you can override to enable you to manage the locking mechanisms and so prevent the corruption of shared data such as the heap. These functions have the prototypes:

_mutex_initialize()

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

int _mutex_initialize(mutex *m);

By default, _mutex_initialize() returns 0 for a nonthreaded application. Therefore, in a multithreaded application, _mutex_initialize() must return a nonzero value on success so that, at runtime, the library knows that it is being used in a multithreaded environment.

_mutex_acquire()

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

void _mutex_acquire(mutex *m);

_mutex_acquire() returns immediately if the mutex has no owner. If the mutex is currently owned by another thread, _mutex_acquire() must block it until it becomes available.

_mutex_release()

This causes the calling thread to release the supplied mutex.

void _mutex_release(mutex *m);

_mutex_release() assumes that the mutex is owned by the calling thread.

For the C library, a mutex is specified as a single 32-bit word of memory that can be placed anywhere. However, if your mutex implementation requires more space than this, or demands that the mutex be in special memory area, then you must treat the default mutex as a pointer to a real mutex.

5.2.5 Using the ARM C libraries with a multithreaded application

In order to use the ARM library in a multithreaded environment, you must provide:

- An implementation of __user_perthread_libspace() that returns a different block of memory for each thread. This can be achieved by either:

  — returning a different address depending on the thread it is called from

  — having a single __user_perthread_libspace block at a fixed address and swapping its contents when switching threads.

  You can use either approach to suit your environment.

  You do not have to re-implement __user_perproc_libspace unless there is a specific reason to do so. In the majority of cases, there is no requirement to re-implement this function.

- Implementations of the three mutex functions (see *Managing locks in multithreaded applications* on page 5-9).

In some applications these functions might not do anything useful. For example, a co-operatively threaded program does not have to take any steps to ensure data integrity, provided it avoids calling its yield function during a critical section. However, in other types of application, for example where you are implementing pre-emptive threading or in a *Symmetric Multi-Processor* (SMP) model, these functions play an important part in handling locks.

• A way to manage multiple stacks.

A simple way to do this is to use the ARM two-region memory model (see *The memory models* on page 5-87). Using this means that you keep the stack that belongs to the primary thread entirely separate from the heap. Then you must allocate further memory for additional stacks from the heap itself.

• Thread management functions, for example, to create or destroy threads, to handle thread synchronization, and to retrieve exit codes.

——— **Note** ———
The ARM C libraries supply no thread management functions of their own and so you must supply any that are required.

• A thread-switching mechanism.

——— **Note** ———
The ARM C libraries supply no thread-switching mechanisms of their own. This is because there are many different ways to do this and the libraries are designed to work with all of them.

When these requirements have been met, you can use the ARM C libraries in your multithreaded environment:

• some functions work independently in each thread

• some functions automatically use the mutex functions to mediate multiple accesses to a shared resource

• some functions are still non-reentrant and so a reentrant equivalent is supplied

• a few functions remain non-reentrant and no alternative is available.

See *Thread-safety in the ARM C libraries* on page 5-12 and *Thread-safety in the ARM C++ libraries* on page 5-18 for full details.

5.2.6   Thread-safety in the ARM C libraries

In the ARM libraries, functions can be thread-safe as follows:

•   Some functions are never thread-safe, for example `setlocale()`.

•   Some functions are always thread-safe, for example `fopen()`.

•   Other functions are only thread-safe if you pass the appropriate arguments, for example `tmpnam()`.

Threading problems might occur when your application makes use of the ARM libraries in a way that is hidden, for example, if you are using the language helper functions. Integer divide uses a helper function because there is no divide instruction in the ARM instruction set. Integer division by zero throws a signal by default, so you must either:

•   re-implement `__rt_raise()` to deal with the signal in a thread-safe manner

•   re-implement `__aeabi_idiv0` so that division by zero returns some standard result (such as `0`) rather than raising a signal.

For details on which C library functions are thread-safe and when, see:
•   *Functions that are thread-safe* on page 5-13
•   *Functions that are not thread-safe* on page 5-16.

Functions that are thread-safe

Table 5-1 shows those C library functions that are thread-safe.

Table 5-1 Functions that are thread-safe

Functions	Description
`calloc()`, `free()`, `malloc()`, `realloc()`	The heap functions are thread-safe. A single heap is shared between all threads, and mutexes are used to avoid data corruption when there is concurrent access. Each heap implementation is responsible for doing its own locking. If you supply your own allocator, it must also do its own locking. This enables it to do fine-grained locking, if required, rather than simply protecting the entire heap with a single mutex (coarse-grained locking).
`__alloca()`, `__alloca_finish()`, `__alloca_init()`, `__alloca_initialize()`	The `alloca` functions are thread-safe. The `alloca` state for each thread is contained in the `__user_perthread_libspace` block. This means that multiple threads do not conflict. ——— **Note** ——— Be aware that the `alloca` functions also use the heap. However, heap functions are all thread-safe.
`abort()`, `raise()`, `signal()`, `fenv.h` functions	The ARM signal handling functions and FP exception traps are thread-safe. The settings for signal handlers and FP traps are global across the entire process and are protected by locks. Data corruption does not occur if multiple threads call `signal()` or an `fenv.h` function at the same time. However, be aware that the effects of the call act on all threads and not just on the calling thread.

Table 5-1 Functions that are thread-safe

Functions	Description
`clearerr()`, `fclose()`, `feof()`, `ferror()`, `fflush()`, `fgetc()`, `fgetpos()`, `fgets()`, `fopen()`, `fputc()`, `fputs()`, `fread()`, `freopen()`, `fseek()`, `fsetpos()`, `ftell()`, `fwrite()`, `getc()`, `getchar()`, `gets()`, `perror()`, `putc()`, `putchar()`, `puts()`, `rewind()`, `setbuf()`, `setvbuf()`,`tmpfile()`, `tmpnam()`, `ungetc()`	The `stdio` library is thread-safe. Each individual stream is protected by a lock, so two threads can each open their own `stdio` stream and use it, without interfering with one another. If two threads both want to read or write the same stream, locking at the `fgetc()` and `fputc()` level prevents data corruption, but it is possible that the individual characters output by each thread might be interleaved in a confusing way. ———— **Note** ———— Be aware that `tmpnam()` also contains a static buffer but this is only used if the argument is NULL. To ensure that your use of `tmpnam()` is thread-safe, supply your own buffer space.
`fprintf()`, `printf()`, `vfprintf()`,`vprintf()`, `fscanf()`, `scanf()`	When using these functions: • The standard C `printf()` and `scanf()` functions use `stdio`, and so are thread-safe. • The standard C `printf()` is susceptible to changes in the locale settings if called in a multithreaded program.
`clock()`	`clock()` contains static data that is written once at program startup and then only ever read. Therefore, `clock()` is thread-safe provided no extra threads are already running at the time that the library is initialized.
`errno()`	`errno` is thread-safe. Each thread has its own `errno` stored in a `__user_perthread_libspace` block. This means that each thread can call `errno`-setting functions independently and then check `errno` afterwards without fearing other threads.
`atexit()`	The list of exit functions maintained by `atexit()` is process-global and protected by a lock. In the worst case, if more than one thread calls `atexit()`, the order that exit functions are called cannot be guaranteed.

Table 5-1 Functions that are thread-safe

Functions	Description
abs(), acos(), asin(), atan(), atan2(), atof(), atol(), atoi(), bsearch(), ceil(), cos(), cosh(), difftime(), div(), exp(), fabs(), floor(), fmod(), frexp(), labs(), ldexp(), ldiv(), log(), log10(), memchr(), memcmp(), memcpy(), memmove(), memset(), mktime(), modf(), pow(), qsort(), sin(), sinh(), sqrt(), strcat(), strchr(), strcmp(), strcpy(), strcspn(), strlen(), strncat(), strncmp(), strncpy(), strpbrk(), strrchr(), strspn(), strstr(), strxfrm(), tan(), tanh()	These functions are thread-safe.
longjmp(), setjmp()	Although setjmp() and longjmp() keep data in __user_libspace, they call the __alloca_* functions, which are thread-safe.
remove(), rename(), time()	These use SWIs that communicate with the ARM debugging environments. Typically, you have to re-implement these for a real-world application.
snprintf(), sprintf(), vsnprintf(), vsprintf(), sscanf(), isalnum(),isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(),ispunct(), isspace(), isupper(), isxdigit(), tolower(), toupper(),strcoll(), strtod(), strtol(),strtoul(), strftime()	When using these functions, the string-based functions read the locale settings. Typically, they are thread-safe. However, if you change locale in mid-session, you should ensure that these functions are not affected. The string-based functions, such as sprintf() and sscanf(), do not depend on the stdio library.
stdin, stdout, stderr	These are thread-safe.

FP status word

The FP status word is safe to use in a multithreaded environment, even in software floating-point. Here, a status word for each thread is stored in its own __user_perthread_libspace block.

——— **Note** ———

Be aware that, in hardware floating-point, the FP status word is stored in a VFP register. In this case, your thread-switching mechanism must keep a separate copy of this register for each thread.

Functions that are not thread-safe

Table 5-2 shows the C library functions that are not thread-safe.

Table 5-2 Functions that are not thread-safe

Functions	Description
setlocale()	The locale settings are global across all threads, and are not protected by a lock. If two threads call setlocale(), there might be data corruption. Also, many other functions, for example strtod() and sprintf(), read the current locale settings, so if one thread calls setlocale() concurrently with another thread calling such a function then there might be unexpected results. ARM recommends that you choose the locale you want and call setlocale() once to initialize it. Do this before creating any additional threads in your program so that any number of threads can read the locale settings concurrently without interfering with one another. Be aware that localeconv() is not thread safe. Call the ARM function _get_lconv() with a pointer to a user-supplied buffer instead.
asctime(), localtime(), strtok()	These functions are all thread-unsafe. Each contains a static buffer that might be overwritten by another thread between a call to the function and the subsequent use of its return value. ARM supplies reentrant versions, asctime_r(), localtime_r(), and strtok_r(), as specified by POSIX. ARM recommends that you use these functions instead to ensure safety. ——— **Note** ——— These reentrant versions take additional parameters that are pointers to a **char** pointer to the next token.
gamma(), lgamma()	These extended mathlib functions, defined in math.h, use a global variable, signgam, and so are not thread-safe. ARM recommends that you use gamma_r() and lgamma_r() instead. These write the sign of the result into memory pointed to by a supplied **int** * argument.

Table 5-2 Functions that are not thread-safe

Functions	Description
mbrlen(), mbsrtowcs(), mbrtowc(), wcrtomb(), wcsrtombs()	The C89 multibyte conversion functions (defined in stdlib.h) are not thread-safe, for example mblen() and mbtowc(), because they contain internal static state that is shared between all threads without locking. However, the extended restartable versions (defined in wchar.h) are thread-safe, for example mbrtowc() and wcrtomb(), provided you pass in a pointer to your own mbstate_t object. You must exclusively use these functions with non-NULL mbstate_t * parameters if you want to ensure thread-safety when handling multibyte strings.
exit()	Do not call exit(), in a multithreaded program, even if you have provided an implementation of the underlying _sys_exit() that actually terminates all threads. In this case, exit() cleans up *before* calling _sys_exit() and so disrupts other threads.
rand(), srand()	These functions keep internal state that is both global and unprotected. This means that calls to rand() are never thread-safe ARM recommends that you use your own locking to ensure that only one thread ever calls rand() at a time, for example by defining $Sub$$rand() if you want to avoid changing your code. Alternatively, do one of the following: • supply your own random number generator that can have multiple independent instances • simply arrange that only one thread ever needs to generate random numbers.

5.2.7   Thread-safety in the ARM C++ libraries

The following list summarizes thread-safety in the C++ libraries:

- The function `std::set_new_handler()` is not thread-safe. This means that some forms of `operator new` and `operator delete` are not thread-safe with respect to `std::set_new_handler()`:

    — The default C++ runtime library implementations of the following use `malloc()` and `free()` and are thread-safe with respect to each other, `malloc()`, and `free()`. They are not thread-safe with respect to `std::set_new_handler()`. You are allowed to replace them.:

        `::operator new(std::size_t)`

        `::operator new[](std::size_t)`

        `::operator new(std::size_t, const std::nothrow_t&)`

        `::operator new[](std::size_t, const std::nothrow_t)`

        `::operator delete(void*)`

        `::operator delete[](void*)`

        `::operator delete(void*, const std::nothrow_t&)`

        `::operator delete[](void*, const std::nothrow_t&)`

    — The following placement forms are also thread-safe. You are not allowed to replace them:

        `::operator new(std::size_t, void*)`

        `::operator new[](std::size_t, void*)`

        `::operator delete(void*, void*)`

        `::operator delete[](void*, void*)`

- Construction and destruction of global objects is not thread-safe.

- Construction of local static objects can be made thread-safe, if you re-implement the functions `__cxa_guard_acquire()`, `__cxa_guard_release()`, `__cxa_guard_abort()`, `__cxa_atexit()` and `__aeabi_atexit()` appropriately. For example, with appropriate re-implementing, the following construction of `lsobj` can be made thread-safe:

    ```
    struct T { T(); };
    void f() { static T lsobj; }
    ```

    The `__cxa_` and `__aeabi_` functions are documented in the *C++ ABI for the ARM Architecture* specification.

- Throwing an exception is thread-safe if any user constructors and destructors that get called, are also thread-safe. See the *Exception Handling ABI for the ARM Architecture* specification.

- The ARM C++ library uses the ARM C library. To use the ARM C++ library in a multi-threaded environment, you must provide the functions described in *Using the ARM C libraries with a multithreaded application* on page 5-10.

5.3 Building an application with the C library

This section covers creating an application that links with functions from the C or C++ libraries. Functions in the C library are responsible for:

- Creating an environment in which a C or C++ program can execute. This includes
  — creating a stack
  — creating a heap, if required
  — initializing the parts of the library the program uses.

- Starting execution by calling `main()`.

- Supporting use of ISO-defined functions by the program.

- Catching runtime errors and signals and, if required, terminating execution on error or program exit.

This section includes:
- *Using the libraries with an application*
- *Building an application for a semihosted environment*
- *Building an application for a nonsemihosted environment* on page 5-22.

5.3.1 Using the libraries with an application

There are three ways to use the libraries with an application:

- Build a semihosted application that can be debugged in a semihosted environment such as with RVI-ME. See *Building an application for a semihosted environment*.

- Build a nonhosted application that, for example, can be embedded into ROM. See *Building an application for a nonsemihosted environment* on page 5-22.

- Build an application that does not use `main()` and does not initialize the library. This application has restricted library functionality, unless you re-implement some functions. See *Building an application without the C library* on page 5-27.

5.3.2 Building an application for 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 typically also 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, RVI-ME

- implementing your own SWI handler for the semihosting SWI (see Chapter 7 *Semihosting*).

See *Overview of semihosting dependencies* on page 5-23 for a list of functions that require semihosting.

It is not necessary to write any new functions or include files if you are using the default semihosting functionality of the library.

Using RealView ICE Micro Edition

RealView ICE Micro Edition (RVI-ME) supports the semihosting SWI, but the memory map assumed by the library might require tailoring to match the hardware being debugged. However, it is easy to tailor the memory map assumed by the C library. See *Tailoring the runtime memory model* on page 5-87.

Using re-implemented functions in a semihosted environment

You can also mix the semihosting functionality with new input/output functions. For example, you can implement fputc() to output directly to hardware such as a UART, in addition to the semihosted implementation. See *Building an application for a nonsemihosted environment* on page 5-22 for information on how to re-implement individual functions.

Converting a semihosted application to a standalone application

After an application has been developed in a semihosted debugging environment, you can move the application to a nonhosted environment by one of the following methods:

- Remove all calls to semihosted functions. See *Avoiding the semihosting SWI* on page 5-24.

- Re-implement the semihosted functions. See *Building an application for a nonsemihosted environment* on page 5-22. You do not have to re-implement all semihosted functions. You must, however, re-implement the functions that you are using in your application.

- Implementing a SWI handler that handles the semihosting SWIs.

Implementing your own semihosting SWI support

It is possible to implement your own semihosting SWI support. The interface is simple and requires a handler for only two SWI numbers:
- 0x123456 (used in ARM state).
- 0xab (used in Thumb state).

See the semihosting SWI definitions in Chapter 7 *Semihosting* and also the include file rt_sys.h for definitions of the functions that call the semihosting SWI.

5.3.3   Building an application for 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.   Add the __use_no_semihosting_swi guard to the source. See *Avoiding the semihosting SWI* on page 5-24.

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, though some C++ functions use underlying C library functions that are target-dependent.

The functions that you might have to re-implement are described in:
* *Tailoring static data access* on page 5-45
* *Tailoring locale and CTYPE* on page 5-46
* *Tailoring error signaling, error handling, and program exit* on page 5-71
* *Tailoring the runtime memory model* on page 5-87
* *Tailoring the input/output functions* on page 5-95
* *Tailoring other C library functions* on page 5-106.

Examples of embedded applications that do not use a hosted environment are included in the main examples directory, in ...\emb_sw_dev.

C++ exceptions in a nonsemihosted environment

The default C++ std::terminate() handler is required by the C++ Standard to call abort(). The default C library implementation of abort() uses functions that require semihosting support. Therefore, if you use exceptions in a nonsemihosted environment, you must provide an alternative implementation of abort().

Overview of semihosting dependencies

Table 5-3 shows those functions that depend directly on semihosting SWIs.

Table 5-3 Direct semihosting SWI dependencies

Function	Description
__user_initial_stackheap()	See *Tailoring the runtime memory model* on page 5-87. You must re-implement this function if you are using scatter-loading.
_sys_exit()_ttywrch()	See *Tailoring error signaling, error handling, and program exit* on page 5-71.
_sys_command_string()_sys_clo se(), _sys_ensure() _sys_iserror(), _sys_istty() _sys_flen(), _sys_open() _sys_read(), _sys_seek() _sys_write(), _sys_tmpnam()	See *Tailoring the input/output functions* on page 5-95.
clock(), _clock_init() remove(), rename()system(), time()	See *Tailoring other C library functions* on page 5-106.

Table 5-4 shows those functions that depend indirectly on one or more of the functions listed in Table 5-3 on page 5-23.

Table 5-4 Indirect semihosting SWI dependencies

Function	Where used
`__raise()`	Catch, handle, or diagnose C library exceptions, without C signal support. See *Tailoring error signaling, error handling, and program exit* on page 5-71.
`__default_signal_handler()`	Catch, handle, or diagnose C library exceptions, with C signal support. See *Tailoring error signaling, error handling, and program exit* on page 5-71.
`__Heap_Initialize()`	Choosing or redefining memory allocation. See *Tailoring storage management* on page 5-77.
`ferror()fputc()__stdout`	Re-implementing the printf family. See *Tailoring the input/output functions* on page 5-95.
`__backspace()fgetc()__stdin`	Re-implementing the scanf family. See *Tailoring the input/output functions* on page 5-95.
`fwrite()fputs()puts()fread()fgets()gets()ferror()`	Re-implementing the stream output family. See *Tailoring the input/output functions* on page 5-95.

Avoiding the semihosting SWI

If you write an application in C, you must link it with the C library even if it makes no direct use of C library functions. The C library contains compiler helper functions and initialization code. Some C library functions use the semihosting SWI.

To avoid using the semihosting SWI, do either of the following:
- re-implement the functions in your own application
- write the application so that it does not call any semihosted function.

To guarantee that no functions using the semihosting SWI are included in your application, use either:
- `IMPORT __use_no_semihosting_swi` from assembly language
- `#pragma import(__use_no_semihosting_swi)` from C.

You need to define this symbol once only in your application. The symbol has no effect except to cause a link-time error if a function that uses the semihosting SWI is included from the library. The linker error message is:

```
Error : L6200E: Symbol __semihosting_swi_guard multiply defined
                (by use_semi.o and use_no_semi.o).
```

Link with `--verbose --errors out.txt`, then search `out.txt` for
`__semihosting_swi_guard`.

API definitions

In addition to the semihosted functions listed in Table 5-3 on page 5-23 and Table 5-4
on page 5-24, Table 5-5 shows functions and files that might be useful when building
for a different environment.

Table 5-5 Published API definitions

File or function	Description
`__main()__rt_entry()`	Initializes the runtime environment and executes the user application.
`__rt_lib_init()__rt_exit()` `__rt_lib_shutdown()`	Initializes or finalizes the runtime library.
`localeCTYPE`	Defines the character properties for the local alphabet. See *Tailoring locale and CTYPE* on page 5-46.
`rt_sys.h`	A C header file describing all the functions whose default (semihosted) implementations use the semihosting SWI.
`rt_heap.h`	A C header file describing the storage management abstract data type.
`rt_locale.h`	A C header file describing the five locale category *filing systems*, and defining some macros that are useful for describing the contents of locale categories.
`rt_misc.h`	A C header file describing miscellaneous unrelated public interfaces to the C library.
`rt_memory.s`	An empty, but commented, prototype implementation of the memory model. See *Writing your own memory model* on page 5-88 for a description of this file.

If you are re-implementing a function that exists in the standard ARM library, the linker
uses an object or library from your project rather than the standard ARM library. A
library you add to a project does not have to follow the ARM library naming convention.

——— **Caution** ———

Do not replace or delete libraries supplied by ARM Limited. You must not overwrite the supplied library files. Place your re-implemented functions in a separate library.

5.4 Building an application without the C library

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

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

- software stack checking
- low-level stdio functions that have the prefix _sys_
- signal-handling functions, signal() and raise() in signal.h
- other functions, such as atexit() and alloca().

See *The standalone C library functions* on page 5-30 for details of the functions that are not available without library initialization.

This section refers to creating applications without the library as *bare machine C*. These applications do not automatically use the full C runtime environment provided by the C library. Even though you are creating an application without the library, some helper functions from the library must be included. There are also many library functions that can be made available with only minor re-implementations.

This section describes:

- *Integer and FP helper functions*
- *Bare machine integer C* on page 5-28
- *Bare machine C with floating-point* on page 5-28
- *Exploiting the C library* on page 5-29
- *The standalone C library functions* on page 5-30.

5.4.1 Integer and FP helper functions

There are several compiler helper functions that are used by the compiler to handle operations that do not have a short machine code equivalent. For example, integer divide uses a helper function because there is no divide instruction in the ARM and Thumb instruction set.

Integer divide, and all the floating-point functions, require __rt_raise() to handle math errors. Re-implementing __rt_raise() enables all the math helper functions, and it avoids having to link in all the signal-handling library code.

5.4.2   Bare machine integer C

If you are writing a program in C that does not use the library and is to run without any environment initialization, you must:

- Implement `__rt_raise()` yourself, because this error-handling function can be called from numerous places within the compiled code.

- Not define `main()` to avoid linking in the library initialization code.

- Not use software stack checking in the build options.

- Write an assembly language veneer that establishes the register state required to run C. This veneer must branch to the entry function in your application.

- Provide your own RW/ZI initialization code.

- Ensure that your initialization veneer is executed by, for example, placing it in your reset handler.

- Build your application using `--fpu none` and link it normally. The linker uses the appropriate C library variant to find any required compiler helper functions.

Many library facilities require `__user_libspace` for static data. Even without the initialization code activated by having a `main()` function, `__user_libspace` is created automatically and uses 96 bytes in the ZI segment. See *The __user_libspace static data area* on page 5-8 for a description of the `__user_libspace` area.

5.4.3   Bare machine C with floating-point

If you want to use floating-point processing in your application you must:

- Perform the steps necessary for integer C as described in *Bare machine integer C*. However, do not build your application with the `--fpu none` option.

- Use the appropriate FPU option when you build your application.

- Call `_fp_init()` to initialize the floating-point status register before performing any floating-point operations.

If you are using software floating-point, you can also define the function `__rt_fp_status_addr()` to return the address of a writable data word to be used instead of the floating-point status register. If you do not do this, the `__user_libspace` area is created, occupying 96 bytes. See *The __user_libspace static data area* on page 5-8 for a description of the `__user_libspace` area.

5.4.4   Exploiting the C library

If you create an application that includes a `main()` function, the linker automatically includes the initialization code necessary for the execution environment. See *Building an application with the C library* on page 5-20 for instructions. There are situations though where this is not desirable or possible.

You can create an application that consists of customized startup 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.

Program design

The functions you must re-implement depend on how much of the library functionality you require:

- If you want only the compiler support functions for division, structure copy, and FP arithmetic, you must provide `__rt_raise()`. This also enables very simple library functions such as those in `errno.h`, `setjmp.h`, and most of `string.h` to work.

- If you call `setlocale()` explicitly, locale-dependent functions are activated. This enables you to use the `atoi` family, `sprintf()`, `sscanf()`, and the functions in `ctype.h`.

- Programs that use floating-point must call `_fp_init()`. If you select software floating-point, the program must also provide `__rt_fp_status_addr()`. If this function is not re-implemented, the default action is to create a `__user_libspace` area. See *The __user_libspace static data area* on page 5-8 for a description of the `__user_libspace` area.

- Implementing high-level input/output support is necessary for functions that use `fprintf()` or `fputs()`. The high-level output functions depend on `fputc()` and `ferror()`. The high-level input functions depend on `fgetc()` and `__backspace()`.

Implementing these functions and the heap enables you to use almost the entire library.

Using low-level functions

If you are using the libraries in an application that does not have a `main()` function, you must re-implement some functions in the library. See *The standalone C library functions* on page 5-30 for a detailed list of functions that are not available, functions that are available without modification, and functions that are available after other lower-level functions are re-implemented.

__rt_raise() is essential. It is required by all FP functions, by integer division (so that divide-by-zero can be reported), and by some other library routines. You probably cannot write a nontrivial program without doing something that requires __rt_raise().

——— **Note** ———

If rand() is called, srand() *must* be called first. This is done automatically during library initialization but not when you avoid the library initialization.

Using high-level functions

High-level I/O functions, fprintf() for example, can be used if the low-level functions, fputc() for example, are re-implemented. Most of the formatted output functions also require a call to setlocale(). See *Tailoring the input/output functions* on page 5-95 for instructions.

Anything that uses locale must not be called before first calling setlocale() to initialize it, for example, call setlocale(LC_ALL, "C"). Locale-using functions are described in *The standalone C library functions*. These include the functions in ctype.h and locale.h, the printf() family, the scanf() family, ato*, strto*, strcoll/strxfrm, and much of time.h.

Using malloc()

If heap support is required for bare machine C, _init_alloc() must be called first to supply initial heap bounds, and __rt_heap_extend() *must* be provided even if it only returns failure. Prototypes for both functions are in rt_heap.h.

5.4.5   The standalone C library functions

The rest of this section lists the include files, and the functions they contain, that are available with an uninitialized library. Some otherwise-unavailable functions can be used if the library functions they depend on are re-implemented.

alloca.h

Functions listed in this file are not available without library initialization. See *Building an application with the C library* on page 5-20 for instructions.

assert.h

Functions listed in this file require high-level stdio, __rt_raise(), and _sys_exit(). See *Tailoring error signaling, error handling, and program exit* on page 5-71 for instructions.

ctype.h

Functions listed in this file require the `locale` functions.

errno.h

Functions in this file work without the requirement for any library initialization or function re-implementation.

fenv.h

Functions in this file work without the requirement for any library initialization and only require the re-implementation of `__rt_raise()`.

float.h

This file does not contain any code. The definitions in the file do not require library initialization or function re-implementation.

inttypes.h

Functions listed in this file require the `locale` functions.

limits.h

Functions in this file work without the requirement for any library initialization or function re-implementation.

locale.h

Call `setlocale()` before calling any function that uses `locale` functions. For example call:

```
setlocale(LC_ALL, "C")
```

See the contents of `locale.h` for details of the following functions and data structures:

setlocale()   Selects the appropriate locale as specified by the category and `locale` arguments.

lconv   Is the structure used by `locale` functions for formatting numeric quantities according to the rules of the current locale.

localeconv() Creates an `lconv` structure and returns a pointer to it.

_get_lconv() Fills the lconv structure pointed to by the parameter. This ISO extension
             removes the requirement for static data within the library.

locale.h also contains constant declarations used with locale functions. See *Tailoring
locale and CTYPE* on page 5-46 for more information.

math.h

For functions in this file to work, you must first call _fp_init() and re-implement
__rt_raise().

setjmp.h

Functions in this file work without any library initialization or function
re-implementation.

signal.h

Functions listed in this file are not available without library initialization. See *Building
an application with the C library* on page 5-20 for instructions on building an
application that uses library initialization.

__rt_raise() can be re-implemented for error and exit handling. See *Tailoring error
signaling, error handling, and program exit* on page 5-71 for instructions.

stdarg.h

Functions listed in this file work without any library initialization or function
re-implementation.

stddef.h

This file does not contain any code. The definitions in the file do not require library
initialization or function re-implementation.

stdint.h

This file does not contain any code. The definitions in the file do not require library
initialization or function re-implementation.

stdio.h

The following dependencies or limitations apply to these functions:

- The high-level functions such as printf(), scanf(), puts(), fgets(), fread(), fwrite(), and perror() depend on lower-level stdio functions fgetc(), fputc(), and __backspace(). You must re-implement these lower-level functions when using the standalone C library.

  However, you cannot re-implement the _sys_ prefixed functions (for example, _sys_read()) when using the standalone C library because they require library initialization.

  See *Tailoring the input/output functions* on page 5-95 for more details.

- The printf() and scanf() family of functions require locale.

- The remove() and rename() functions are system-specific and probably not usable in your application.

stdlib.h

Most functions in this file work without any library initialization or function re-implementation. The following functions depend on other functions being instantiated correctly:

ato*()	Requires locale.
strto*()	Requires locale.
malloc()	malloc(), calloc(), realloc(), and free() require heap functions.
atexit()	This is not available when building an application without the C library.

string.h

Functions in this file work without any library initialization, with the exception of strcoll() and strxfrm(), that require locale.

time.h

mktime() and localtime() can be used immediately.

time() and clock() are system-specific and probably not usable unless re-implemented.

asctime(), ctime(), and strftime() require locale.

wchar.h

Wide character library functions added to ISO C by *Normative Addendum 1* in 1994.

- Support for wide-character output and format strings, swprintf(), vswprintf(), swscanf(), and vswscanf().

- All the conversion functions (for example, btowc, wctob, mbrtowc, and wcrtomb) require locale.

- wcscoll and wcsxfrm require locale.

wctype.h

Wide character library functions added to ISO C by *Normative Addendum 1* in 1994. This requires locale.

5.5 Tailoring the C library to a new execution environment

This section describes how to re-implement functions to produce an application for a different execution environment, for example, embedded in ROM or used with a *Real Time Operating System* (RTOS).

Symbols that have a single or double underscore, _ or __, name functions that are used as part of the low-level implementation. You can re-implement some of these functions.

Additional information on these library functions is available in the rt_heap.h, rt_locale.h, rt_misc.h, and rt_sys.h include files and the rt_memory.s assembler file.

Also, see the following specifications for descriptions of functions that have the prefix __cxa or __aeabi:

* *C Library ABI for the ARM Architecture*
* *Exception Handling ABI for the ARM Architecture*
* *C++ ABI for the ARM Architecture*.

This section includes:

* *How C and C++ programs use the library functions*
* *__rt_entry* on page 5-41
* *Exiting from the program* on page 5-41
* *__rt_exit()* on page 5-42
* *__rt_lib_init()* on page 5-43
* *__rt_lib_shutdown()* on page 5-44.

5.5.1 How C and C++ programs use the library functions

This section describes:

* specific library functions that are used to initialize the execution environment and application

* library exit functions

* target-dependent library functions that the application itself might call during its execution.

Initializing the execution environment and executing the application

The entry point of a program is at __main in the C library where library code does the following:

1. Copies nonroot (RO and RW) execution regions from their load addresses to their execution addresses. Also, if any data sections are compressed, they are decompressed from the load address to the execution address. See *RealView Developer Kit v2.2 Linker and Utilities Guide* for more details.

2. Zeroes ZI regions.

3. Branches to __rt_entry.

If you do not want the library to perform these actions, you can define your own __main that branches to __rt_entry as shown in Example 5-1.

Example 5-1 __main and __rt_entry

```
    IMPORT __rt_entry
    EXPORT __main
    ENTRY
__main
    B       __rt_entry
    END
```

The library function __rt_entry() runs the program as follows:

1. Calls __rt_stackheap_init() to set up the stack and heap.

2. Calls __rt_lib_init() to initialize referenced library functions, initialize the locale and, if necessary, set up argc and argv for main().

    For C++, calls the constructors for any top-level objects by way of __cpp_initialize__aeabi_. See *C++ initialization, construction, and destruction* on page 5-37 for more details.

3. Calls main(), the user-level root of the application.

    From main(), your program might call, among other things, library functions. See *Library functions called from main()* on page 5-40 for more information.

4. Calls exit() with the value returned by main().

C++ initialization, construction, and destruction

C++ places certain requirements on the construction and destruction of objects with static storage duration (see *section 3.6 of the C++ Standard*).

The ARM C++ compiler uses the `.init_array` area to achieve this. The `.init_array` area is internal to the RVCT tools, and is handled automatically by the tools. Because this area is internal to RVCT there is no guarantee that it might not change between releases. The following details might help to explain its role.

The `.init_array` area is a const data array of self-relative pointers to functions. For example, you might have the following C++ translation unit, contained in the file `test.cpp`:

```
struct T { T(); ~T(); } t;
int f() { return 4; }
int i = f();
```

This translates into the following pseudo-code:

```
        AREA ||.text||, CODE, READONLY
    int f() { return 4; }
    static void __sti___8_test_cpp {
        // construct 't' and register its destruction
        __aeabi_atexit(T::T(&t), &T::~T, &__dso_handle);
        i = f();
    }
        AREA ||.init_array||, DATA, READONLY, ALIGN=2
        DCD __sti___8_test_cpp - {PC}
        AREA ||.data||, DATA, ALIGN=2
    t   % 4
    i   % 4
```

This pseudo-code is for illustration only.

The linker collects each `.init_array` from the various translation units together. It is important that the `.init_array` is accumulated in the same order.

The library routine `__cpp_initialize__aeabi_` is called from the C library startup code (`__rt_lib_init`) before main. `__cpp_initialize__aeabi_` walks through the `.init_array` calling each function in turn. On exit, `__rt_lib_shutdown` calls `__cxa_finalize`.

Usually there is at most one function for `T::T()` (mangled name `_ZN1TC1Ev`), one function for `T::~T()` (mangled name `_ZN1TD1Ev`), one `__sti__` function, and four bytes of `.init_array` for each translation unit. The mangled name for the function `f()` is `_Z1fv`. There is no way to determine the initialization order between translation units.

Function-local static objects with destructors are also handled using `__cxa_atexit`.

Certain sections must be placed contiguously within the same region, for their base/limit symbols to be accessible. If they are not, the linker generates an error like:

```
L6216E: Cannot use base/limit symbols for non-contiguous section .init_array
```

For example, the `.init_array` area is generated by the C++ compiler. This area must be placed in the same region as `__cpp_initialize__aeabi_`. To do this, specify the sections in a scatter-loading description file, for example:

```
LOAD_ROM 0x0000000
{
    EXEC_ROM 0x0000000
    {
        init_aeabi.o(+RO)
        * (.init_array)
        ...
    }
    RAM 0x0100000
    {
        * (+RW,+ZI)
    }
}
```

Legacy support

In RVCT v2.1, or later, `.init_array` replaces `C$$pi_ctorvec`. However, objects with `C$$pi_ctorvec` are still supported. Therefore, if you have legacy objects, your scatter file contains:

```
LOAD_ROM 0x0000000
{
    EXEC_ROM 0x0000000
    {
        init_aeabi.o(+RO)
        init.o(+RO)              ; backwards compatibility
        * (.init_array)
        * (C$$pi_ctorvec)        ; backwards compatibility
        ...
    }
    RAM 0x0100000
    {
        * (+RW,+ZI)
    }
}
```

In RVCT, the C++ name mangling scheme is different from that in ADS v1.2. This might have an effect when loading a newer C++ image into an older debugger.

Exceptions system initialization

The exceptions system can be initialized either on demand (that is, when first used), or before main is entered. Initialization on demand has the advantage of not allocating heap memory unless the exceptions system is used, but has the disadvantage that it becomes impossible to throw any exception (such as std::bad_alloc) if the heap is exhausted at the time of first use.

The default is to initialize on demand. To initialize the exceptions system before main is entered, include the following function in the link:

```
extern "C" void __cxa_get_globals(void);
extern "C" void __ARM_exceptions_init(void)
{
    __cxa_get_globals();
}
```

Although you can place the call to __cxa_get_globals directly in your code, placing it in __ARM_exceptions_init ensures that it is called as early as possible. That is, before any global variables are initialized and before main is entered.

__ARM_exceptions_init is weakly referenced by the library initialization mechanism, and is called if it is present as part of __rt_lib_init.

——— **Note** ———

The exception system is initialized by calls to various library functions, for example, std::set_terminate(). Therefore, you might not have to initialize before the entry to main.

Emergency buffer memory for exceptions

You can choose whether or not to allocate emergency memory that is to be used for throwing a std::bad_alloc exception when the heap is exhausted.

To allocate emergency memory, you must include the symbol __ARM_exceptions_buffer_required in the link. A call is then made to __ARM_exceptions_buffer_init() as part of the exceptions system initialization. The symbol is not included by default.

The following routines manage the exceptions emergency buffer:

`extern "C" void *__ARM_exceptions_buffer_init()`

> Called at most once by the runtime, to allocate the emergency buffer memory. It returns a pointer to the emergency buffer memory, or NULL if no memory is allocated.

`extern "C" void *__ARM_exceptions_buffer_allocate(void *buffer, size_t size)`

> Called when an exception is about to be thrown, but there is not enough heap memory available to allocate the exceptions object. *buffer* is the value previously returned by __ARM_exceptions_buffer_init(), or NULL if that routine was not called. __ARM_exceptions_buffer_allocate() returns a pointer to *size* bytes of memory that is aligned on an eight-byte boundary, or NULL if the allocation is not possible.

`extern "C" void *__ARM_exceptions_buffer_free(void *buffer, void *addr)`

> Called to free memory possibly allocated by __ARM_exceptions_buffer_allocate(). *buffer* is the value previously returned by __ARM_exceptions_buffer_init(), or NULL if that routine was not called. The routine determines whether the passed address has been allocated from the emergency memory buffer, and if so, frees it appropriately, then returns a non-NULL value. If the memory at *addr* was not allocated by __ARM_exceptions_buffer_allocate(), the routine must return NULL.

Default definitions of these routines are present in the image, but you can supply your own versions to override the defaults supplied by the library. The default routines reserve just enough space for a single std::bad_alloc exceptions object. If you do not require an emergency buffer, it is safe to redefine all these routines to return only NULL.

Library functions called from main()

The function main() is the user-level root of the application. It requires the execution environment to be initialized, and that input/output functions can be called. While in main() the program might perform one of the following actions that calls user-customizable functions in the C library:

-

-

-

- Perform floating-point calculations that require the fpu or floating-point library.

- Input or output directly through low-level functions, for example putc(), or indirectly through high-level input/output functions and input/output support functions, for example fprintf() or sys_open(). See *Tailoring the input/output functions* on page 5-95.

- Raise an error or other signal, for example ferror. See *Tailoring error signaling, error handling, and program exit* on page 5-71.

5.5.2   __rt_entry

The symbol __rt_entry is the starting point for a program using the ARM C library. Control passes to __rt_entry after all scatter-load regions have been relocated to their execution addresses.

Implementation

The default implementation of __rt_entry:
1.   Sets up the heap and stack.
2.   Initializes the C library, by calling __rt_lib_init.
3.   Calls main().
4.   Shuts down the C library, by calling __rt_lib_shutdown.
5.   Exits.

__rt_entry must end with a call to one of the following functions:

exit()   Calls atexit()-registered functions and shuts down the library.

__rt_exit() Correctly shuts down the library but does not call atexit() functions.

_sys_exit() Exits directly to the execution environment. It does not shut down the library and does not call atexit() functions. See *_sys_exit()* on page 5-72.

5.5.3   Exiting from the program

The program can exit normally at the end of main() or it can exit prematurely because of an error.

See also:
- *__rt_entry*
- *__rt_exit()* on page 5-42
- *Tailoring error signaling, error handling, and program exit* on page 5-71.

Exiting from an assert

The exit sequence from an assert is:

1. assert() prints a message on stderr.

2. assert() calls abort().

3. abort() calls __rt_raise().

4. If __rt_raise() returns, abort() tries to finalize the library.

If you are creating an application that does not use the library, assert() works if you re-implement abort() and the stdio functions.

One solution for retargeting is to re-implement the assert() function itself. The function prototype is:

void __assert(**const char** *expr*, **const char** *file*, **int** *line*);

where:
*   *expr* points to the string representation of the expression that was not TRUE
*   *file* and *line* identify the source location of the assertion.

The default behavior for assert() depends on the optimization level you are using:

*   If you are using -O0 or -O1, assert() prints a message on stderr.

*   If you are using a higher level of optimization, that is -O2 or -O3, assert() calls abort() without displaying a message.

*   If you are using a higher level of optimization, that is -O2 or -O3, re-implementing the assert() function does not work as described.

You can restore the default behavior for assert() at higher optimization levels by defining __ASSERT_MSG.

5.5.4   __rt_exit()

This function shuts down the library but does not call functions registered with atexit().

Syntax

void __rt_exit(**int** *code*)

code   Is not used by the standard function.

Implementation

Shuts down the C library by calling __rt_lib_shutdown, and then calls _sys_exit to terminate the application. Re-implement _sys_exit rather than __rt_exit, see *_sys_exit()* on page 5-72 for details.

Returns

The function does not return.

5.5.5   __rt_lib_init()

This is the library initialization function and is the companion to __rt_lib_shutdown().

Syntax

extern value_in_regs **struct** __argc_argv __rt_lib_init(**unsigned** *heapbase*, **unsigned** *heaptop*)

where:

heapbase   The start of the heap memory block.

heaptop   The end of the heap memory block.

Implementation

This is the library initialization function. It is called immediately after __rt_stackheap_init() and passed an initial chunk of memory to use as a heap. This function is the standard ARM library initialization function and must not be re-implemented.

Returns

The function returns argc and argv ready to be passed to main(). The structure is returned in the registers as:

```
struct __argc_argv {    int argc;    char **argv;
    int r2, r3;};
```

5.5.6   __rt_lib_shutdown()

This is the library shutdown function and is the companion to __rt_lib_init().

Syntax

void __rt_lib_shutdown(**void**)

Implementation

This is the library shutdown function and is provided in case a user must call it directly. This is the standard ARM library shutdown function and must not be re-implemented.

5.6 Tailoring static data access

This section describes using callouts from the C library to access static data. C library functions that use static data can be categorized as follows:

- functions that do not use any static data of any kind, for example fprintf()

- functions that manage a static state, such as malloc(), rand(), and strtok()

- functions that do not manage a static state, but use static data in a way that is specific to the implementation in the ARM compiler, for example isalpha().

When the C library does something that requires implicit static data, it uses a callout to a function you can replace. These functions are shown in Table 5-6. They do not use semihosting.

Table 5-6 Callouts

Function	Description
__rt_errno_addr()	Called to get the address of the variable errno. See *__rt_errno_addr()* on page 5-73.
__rt_fp_status_addr()	Called by the floating-point support code to get the address of the floating-point status word. See *__rt_fp_status_addr()* on page 5-76.
locale functions	The function __user_libspace() creates a block of private static data for the library. See *Tailoring locale and CTYPE* on page 5-46, and *Writing reentrant and thread-safe code* on page 5-6.

See also *Tailoring the runtime memory model* on page 5-87 for more information about memory use.

The default implementation of __user_libspace creates a 96-byte block in the ZI segment. Even if your application does not have a main() function, the __user_libspace() function does not normally have to be redefined. However, if you are writing an operating system or a process switcher, you must re-implement this function (see *Writing reentrant and thread-safe code* on page 5-6).

—— **Note** ——

Exactly which functions use static data in their definitions might change in future releases.

5.7 Tailoring locale and CTYPE

This section describes functions related to locale. Applications use locale when they display or process data that is dependent on the local language or region, for example character order, monetary symbols, decimal point, time, and date.

See the `rt_locale.h` include file for more information on locale-related functions.

This section includes:

5.7.1 Selecting locale at link time

The `locale` subsystem of the C library can be selected at link time or extended to be selectable at runtime. The following points describe the use of locale categories by the library:

- The default implementation of each locale category is for the C locale. The library also provides an alternative, ISO8859-1 (Latin 1 alphabet) implementation of each locale category that you can select at link time.

- Both the C and ISO8859-1 default implementations provide only one locale to select at runtime.

- You can replace each locale category individually.

- You can include as many locales in each category as you choose and you can name your locales as you choose.

- Each locale category uses one word in the private static data of the library.

- The locale category data is read-only and position independent.

- scanf() forces the inclusion of the LC_CTYPE locale category, but in either of the default locales this adds only 260 bytes of read-only data to several kilobytes of code.

To select the required locale, use either:
- IMPORT __use_iso8859_ctype from assembly language
- #pragma import(__use_iso8859_ctype) from C.

For implementation details, see:
- *ISO8859-1 Implementation*
- *Shift-JIS and UTF-8 Implementation* on page 5-48.

ISO8859-1 Implementation

Table 5-7 shows the ISO8859-1 (Latin-1 alphabet) locale categories.

Table 5-7 Default ISO8859-1 locales

Symbol	Description
__use_iso8859_ctype	Selects the ISO8859-1 (Latin-1) classification of characters (this is essentially 7-bit ASCII, except that the top-bit-set character codes 160-255 represent a selection of useful European punctuation characters, letters, and accented letters).
__use_iso8859_collate	Selects the strcoll/strxfrm collation table appropriate to the Latin-1 alphabet. The default C locale does not require a collation table.
__use_iso8859_monetary	Selects the Sterling monetary category using Latin-1 coding.
__use_iso8859_numeric	Selects separating thousands with commas in the printing of numeric values.
__use_iso8859_locale	Selects all the ISO8859-1 selections described in this table.

There is no ISO8859-1 version of the LC_TIME category.

The C library tests for the existence of the callout function before calling it. If the function does not exist, a default action is taken.

Shift-JIS and UTF-8 Implementation

Table 5-8 shows the Shift-JIS (Japanese characters) or UTF-8 (Unicode characters) locale categories.

Table 5-8 Default Shift-JIS and UTF-8 locales

Function	Description
`__use_sjis_ctype()`	Sets the character set to the Shift-JIS multibyte encoding of Japanese characters.
`__use_utf8_ctype()`	Sets the character set to the UTF-8 multibyte encoding of all Unicode characters.

The following points describe the effects of Shift-JIS encoding:

- The ordinary ctype functions behave correctly on any byte value that is a self-contained character in Shift-JIS. For example, the half-width katakana, which Shift-JIS encodes as single bytes between `0xA6` and `0xDF`, is treated as alphabetic by `isalpha()`.

- The multibyte conversion functions, such as `mbrtowc()`, `mbsrtowcs()`, and `wcrtomb()`, all convert between wide strings in Unicode and multibyte character strings in Shift-JIS.

- `printf("%ls")` expects to convert a Unicode wide string into Shift-JIS output, and `scanf("%ls")` expects to convert Shift-JIS input into a Unicode wide string.

You can arbitrarily switch between multibyte locales and single-byte locales at runtime. For example, you can switch from Shift-JIS to UTF-8 to ISO8859-1 to vanilla C, and back again. By default, only one locale at a time is included.

5.7.2 Selecting locale at runtime

The C library function `setlocale()` selects a locale at runtime for the locale category, or categories, specified in its arguments. It does this by selecting the requested `locale` separately in each locale category. In effect, each locale category is a small filing system containing an entry for each `locale`.

Each locale category is processed by a function like `_get_lc_category`, for example:

```
void const *_get_lc_time (void *null, char const *locale_name)
```

_get_lc_time() returns the address of the time filing system entry for the locale named locale_name, or NULL if the entry was not found.

The implementation of each locale category must supply a selection function as shown in Table 5-9.

Table 5-9 Locale categories

Function	Description
_get_lc_ctype()	Returns a pointer to the first element in a user-defined array that holds character attributes. See *_get_lc_ctype()* on page 5-51.
_get_lc_collate()	Returns a pointer to the first element in a user-defined array that holds sorting attributes. See *_get_lc_collate()* on page 5-53.
_get_lc_monetary()	Returns a pointer to the user-defined __lc_monetary_blk structure. See *_get_lc_monetary()* on page 5-57.
_get_lc_numeric()	Returns a pointer to the user-defined __lc_numeric_blk structure. See *_get_lc_numeric()* on page 5-58.
_get_lc_time()	Returns a pointer to the user-defined __lc_time_blk structure. See *_get_lc_time()* on page 5-59.

C header files describing what must be implemented, and providing some useful support macros, are given in locale.h and rt_locale.h.

Implementation

For each category, changing locale is achieved by changing a pointer into the read-only data for the locale category. Except for default locales, the data must be user-supplied.

All locale blocks for a category are collected into a read-only, position-independent, in-memory file system structure. The C library provides a set of macros to create the blocks and the _findlocale() function to search the file system.

You can define a set of runtime selectable locales by using the supplied re-implementations as a starting point. Your application does not call _get_lc_*category* functions directly. _get_lc_*category* functions are called by setlocale() and __rt_lib_init(). You implement new locales by providing new locale definition blocks and re-implementations of _get_lc_*category* for setlocale() to use as in Example 5-2 on page 5-50.

Example 5-2 get_lc_ctype

```
void const *_get_lc_ctype(void const *null, char const *name) {
    return _findlocale(&lcctype_c_index, name);
}
```

5.7.3   Macros and utility functions

The macros and utility functions listed in Table 5-10 simplify the process of creating and using locale blocks. See the rt_locale.h file for more information.

Table 5-10 locale macros

Function or macro	Description
__LC_CTYPE_DEF	Use this macro to create a block of values for the character set. See *_get_lc_ctype()* on page 5-51.
__LC_COLLATE_DEF	Use this macro to create a block of sorting values for the character set. See *_get_lc_collate()* on page 5-53.
__LC_TIME_DEF	Use this macro to create a block of time formatting values. See *_get_lc_time()* on page 5-59.
__LC_NUMERIC_DEF	Use this macro to create a block of numeric formatting values. See *_get_lc_numeric()* on page 5-58.
__LC_MONETARY_DEF	Use this macro to create a block of monetary formatting values. See *_get_lc_monetary()* on page 5-57.
__LC_INDEX_END	Use this macro to declare the end of an index of formatting values. See *Using the macros*.
_findlocale()	Use this function to return the address of a locale block. See *_findlocale()* on page 5-63.

Using the macros

The data blocks for a single locale category must be contiguous and the LC_INDEX_END macro must be the last macro in the sequence.

The examples in each locale category use two test macros that are defined as:

```
#define EQI(i,j) assert(i==j)
#define EQS(s,t) assert(!strcmp(s,t))
```

5.7.4 _get_lc_ctype()

The CTYPE implementation is selected at link time to be one of the following:

- The C locale only. This is the default.
- The ISO8859-1 (Latin-1) locale.
- Shift-JIS.
- UTF-8.

You can define your own CTYPE attribute table with the following characteristics:

- It must be read-only.

- It is a byte array with indexes ranging from –1 to 255 inclusive (257 bytes in total)

- Each byte is interpreted as eight attribute bits. The values are defined in ctype.h as follows:

__S	white-space characters
__P	punctuation characters
__B	blank characters
__L	lowercase letters
__U	uppercase letters
__N	decimal digits
__C	control characters
__X	hexadecimal-digit letters A-F and a-f
__A	alphabetic but neither uppercase nor lowercase, such as Japanese katakana.

The first element in the array, the element located at –1, must be zero. A skeletal implementation of the functions that return CTYPE data is shown in Example 5-3. There are also macros that define multibyte LC_CTYPE locales, for example, LC_MBCTYPE_DEF. See the file rt_locale.h for details.

Example 5-3 LC_CTYPE_DEF Table

```
__LC_CTYPE_DEF(lcctype_c, "C")
{
    __C, __C, __C, __C, __C, __C, __C, __C, __C,              /* 0x00-0x08 */
    __C+__S,__C+__S,__C+__S,__C+__S,__C+__S,     /* 0x09-0x0D (BS,LF,VT,FF,CR) */
    __C, __C, __C, __C, __C, __C, __C, __C, __C,              /* 0x0E-0x16 */
    __C, __C, __C, __C, __C, __C, __C, __C, __C,              /* 0x17-0x1F */
    __B+__S,                                                  /* space */
```

```
    __P, __P, __P, __P, __P, __P, __P, __P,                /* !"#$%&'( */
    __P, __P, __P, __P, __P, __P, __P,                     /* )*+,-./ */
    __N, __N, __N, __N, __N, __N, __N, __N, __N, __N,       /* 0-9 */
    __P, __P, __P, __P, __P, __P, __P,                     /* :;<=>?@ */
    __U+__X, __U+__X, __U+__X, __U+__X, __U+__X, __U+__X,   /* A-F */
    __U, __U, __U, __U, __U, __U, __U, __U, __U, __U,       /* G-P */
    __U, __U, __U, __U, __U, __U, __U, __U, __U, __U,       /* Q-Z */
    __P, __P, __P, __P, __P, __P,                           /* [\]^_` */
    __L+__X, __L+__X, __L+__X, __L+__X, __L+__X, __L+__X,   /* a-f */
    __L, __L, __L, __L, __L, __L, __L, __L, __L, __L,       /* g-p */
    __L, __L, __L, __L, __L, __L, __L, __L, __L, __L,       /* q-z */
    __P, __P, __P, __P,                                     /* {|}~ */
    __C,                                                    /* 0x7F */
    /* the whole of the top half is illegal characters */
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
};
__LC_CTYPE_DEF(lcctype_iso8859_1, "ISO8859-1")
{
    __C, __C, __C, __C, __C, __C, __C, __C, __C,           /* 0x00-0x08 */
    __C+__S,__C+__S,__C+__S,__C+__S,__C+__S,               /* 0x09-0x0D (BS,LF,VT,FF,CR) */
    __C, __C, __C, __C, __C, __C, __C, __C, __C,           /* 0x0E-0x16 */
    __C, __C, __C, __C, __C, __C, __C, __C, __C,           /* 0x17-0x1F */
    __B+__S,                                               /* space */
    __P, __P, __P, __P, __P, __P, __P, __P,                /* !"#$%&'( */
    __P, __P, __P, __P, __P, __P, __P,                     /* )*+,-./ */
    __N, __N, __N, __N, __N, __N, __N, __N, __N, __N,       /* 0-9 */
    __P, __P, __P, __P, __P, __P, __P,                     /* :;<=>?@ */
    __U+__X, __U+__X, __U+__X, __U+__X, __U+__X, __U+__X,   /* A-F */
    __U, __U, __U, __U, __U, __U, __U, __U, __U, __U,       /* G-P */
    __U, __U, __U, __U, __U, __U, __U, __U, __U, __U,       /* Q-Z */
    __P, __P, __P, __P, __P, __P,                           /* [\]^_` */
    __L+__X, __L+__X, __L+__X, __L+__X, __L+__X, __L+__X,   /* a-f */
    __L, __L, __L, __L, __L, __L, __L, __L, __L, __L,       /* g-p */
    __L, __L, __L, __L, __L, __L, __L, __L, __L, __L,       /* q-z */
    __P, __P, __P, __P,                                     /* {|}~ */
    __C,                                                    /* 0x7F */
    /* ISO8859-1 top half:
     * - 0x80-0x9f are control chars
     * - 0xa0 is nonbreaking space (whitespace)
     * - 0xa1-0xbf are punctuation chars
     * - 0xc0-0xdf are uppercase chars except times sign at 0xd7
     * - 0xe0-0xff are lowercase chars except divide sign at 0xf7 */
    __C,__C,__C,__C,__C,__C,__C,__C,        /* 0x80 - 0x87 */
```

```
            __C,__C,__C,__C,__C,__C,__C,__C,        /* 0x88 - 0x8f */
            __C,__C,__C,__C,__C,__C,__C,__C,        /* 0x90 - 0x97 */
            __C,__C,__C,__C,__C,__C,__C,__C,        /* 0x98 - 0x9f */
            __B+__S,__P,__P,__P,__P,__P,__P,__P,    /* 0xa0 - 0xa7 */
            __P,__P,__P,__P,__P,__P,__P,__P,        /* 0xa8 - 0xaf */
            __P,__P,__P,__P,__P,__P,__P,__P,        /* 0xb0 - 0xb7 */
            __P,__P,__P,__P,__P,__P,__P,__P,        /* 0xb8 - 0xbf */
            __U,__U,__U,__U,__U,__U,__U,__U,        /* 0xc0 - 0xc7 */
            __U,__U,__U,__U,__U,__U,__U,__U,        /* 0xc8 - 0xcf */
            __U,__U,__U,__U,__U,__U,__U,__P,        /* 0xd0 - 0xd7 */
            __U,__U,__U,__U,__U,__U,__U,__U,        /* 0xd8 - 0xdf */
            __L,__L,__L,__L,__L,__L,__L,__L,        /* 0xe0 - 0xe7 */
            __L,__L,__L,__L,__L,__L,__L,__L,        /* 0xe8 - 0xef */
            __L,__L,__L,__L,__L,__L,__L,__P,        /* 0xf0 - 0xf7 */
            __L,__L,__L,__L,__L,__L,__L,__L,        /* 0xf8 - 0xff */
};
_LC_INDEX_END(lcctype_dummy)
void const *_get_lc_ctype(void const *null, char const *name) {
    return _findlocale(&lcctype_c_index, name);
}
void test_lc_ctype(void) {
    EQS(setlocale(LC_CTYPE, NULL), "C"); /* verify starting point */
    EQI(!!isalpha('@'), 0);                 /* test off-by-one */
    EQI(!!isalpha('A'), 1);
    EQI(!!isalpha('\xc1'), 0);              /* C locale: isalpha(Aacute)==0 */
    EQI(!setlocale(LC_CTYPE, "ISO8859-1"), 0);   /* setlocale should work */
    EQS(setlocale(LC_CTYPE, NULL), "ISO8859-1");
    EQI(!!isalpha('@'), 0);                 /* test off-by-one */
    EQI(!!isalpha('A'), 1);
    EQI(!!isalpha('\xc1'), 1);              /* ISO8859 locale: isalpha(Aacute)!=0
*/
    EQI(!setlocale(LC_CTYPE, "C"), 0);   /* setlocale should work */
    EQS(setlocale(LC_CTYPE, NULL), "C");
    EQI(!!isalpha('@'), 0);                 /* test off-by-one */
    EQI(!!isalpha('A'), 1);
    EQI(!!isalpha('\xc1'), 0);              /* C locale: isalpha(Aacute)==0 */
}
```

5.7.5   _get_lc_collate()

_get_lc_collate() must return a pointer to the 0th entry in an array of unsigned bytes whose indexes range from 0 to 255 inclusive (256 bytes total).

Each element gives the position in the collation sequence of the character represented by the index of the element. For example, if you want strcoll() to sort strings beginning with Z in between those beginning with A and those beginning with B, you can set up the LC_COLLATE table so that array['A'] < array['Z'] and array['Z'] < array['B'].

_get_lc_collate() must return a pointer to a collate structure. Use the macros in Example 5-4 to create the structure.

Example 5-4 LC_COLLATE_DEF Table

```
__LC_COLLATE_TRIVIAL_DEF(lccoll_c, "C")
__LC_COLLATE_DEF(lccoll_iso8859_1, "ISO8859-1")
{
    /* Things preceding letters have normal ASCII ordering */
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
    0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
    0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f,
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
    0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f,
    0x40,  /* @ */    0x41,  /* A - then 7 A variants */
    0x49,  /* B */    0x4a,  /* C - then 1 C variant */
    0x4c,  /* D */    0x4d,  /* E - then 4 E variants */
    0x52,  /* F */    0x53,  /* G */
    0x54,  /* H */    0x55,  /* I - then 4 I variants */
    0x5a,  /* J */    0x5b,  /* K */
    0x5c,  /* L */    0x5d,  /* M */
    0x5e,  /* N - then 1 N variant */
    0x60,  /* O - then 6 O variants */
    0x67,  /* P */    0x68,  /* Q */
    0x69,  /* R */    0x6a,  /* S */
    0x6b,  /* T */    0x6c,  /* U - then 4 U variants */
    0x71,  /* V */    0x72,  /* W */
    0x73,  /* X */    0x74,  /* Y - then 1 Y variant */
    0x76,  /* Z - then capital Eth & Thorn */
    0x79,  /* [ */    0x7a,  /* \ */
    0x7b,  /* ] */    0x7c,  /* ^ */
    0x7d,  /* _ */    0x7e,  /* ` */
    0x7f,  /* a - then 7 a variants */
    0x87,  /* b */    0x88,  /* c - then 1 c variant */
    0x8a,  /* d */    0x8b,  /* e - then 4 e variants */
    0x90,  /* f */    0x91,  /* g */
    0x92,  /* h */    0x93,  /* i - then 4 i variants */
    0x98,  /* j */    0x99,  /* k */
    0x9a,  /* l */    0x9b,  /* m */
    0x9c,  /* n - then 1 n variant */
    0x9e,  /* o - then 6 o variants */
    0xa5,  /* p */    0xa6,  /* q */
    0xa7,  /* r */    0xa8,  /* s - then 1 s variant */
    0xaa,  /* t */    0xab,  /* u - then 4 u variants */
```

```
            0xb0,  /* v */    0xb1,  /* w */
            0xb2,  /* x */    0xb3,  /* y - then 2 y variants */
            0xb6,  /* z - then eth & thorn */
            0xb9,  /* { */    0xba,  /* | */
            0xbb,  /* } */    0xbc,  /* ~ */
            0xbd,  /* del */
            /* top bit set control characters */
            0xbe, 0xbf, 0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5,
            0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xcb, 0xcc, 0xcd,
            0xce, 0xcf, 0xd0, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5,
            0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd,
            /* other non_alpha */
            0xde, 0xdf, 0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5,
            0xe6, 0xe7, 0xe8, 0xe9, 0xea, 0xeb, 0xec, 0xed,
            0xee, 0xef, 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5,
            0xf6, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd,
            0x42,  /* A grave */     0x43,  /* A acute */
            0x44,  /* A circumflex */
            0x45,  /* A tilde */     0x46,  /* A umlaut */
            0x47,  /* A ring */      0x48,  /* AE */
            0x4b,  /* C cedilla */  0x4e,  /* E grave */
            0x4f,  /* E acute */     0x50,  /* E circumflex */
            0x51,  /* E umlaut */    0x56,  /* I grave */
            0x57,  /* I acute */     0x58,  /* I circumflex */
            0x59,  /* I umlaut */    0x77,  /* Eth */
            0x5f,  /* N tilde */     0x61,  /* O grave */
            0x62,  /* O acute */     0x63,  /* O circumflex */
            0x64,  /* O tilde */     0x65,  /* O umlaut */
            0xfe,  /* multiply */    0x66,  /* O with line */
            0x6d,  /* U grave */     0x6e,  /* U acute */
            0x6f,  /* U circumflex */ 0x70,  /* U umlaut */
            0x75,  /* Y acute */     0x78,  /* Thorn */
            0xa9,  /* german sz */  0x80,  /* a grave */
            0x81,  /* a acute */     0x82,  /* a circumflex */
            0x83,  /* a tilde */     0x84,  /* a umlaut */
            0x85,  /* a ring */      0x86,  /* ae */
            0x89,  /* c cedilla */  0x8c,  /* e grave */
            0x8d,  /* e acute */     0x8e,  /* e circumflex */
            0x8f,  /* e umlaut */    0x94,  /* i grave */
            0x95,  /* i acute */     0x96,  /* i circumflex */
            0x97,  /* i umlaut */    0xb7,  /* eth */
            0x9d,  /* n tilde */     0x9f,  /* o grave */
            0xa0,  /* o acute */     0xa1,  /* o circumflex */
            0xa2,  /* o tilde */     0xa3,  /* o umlaut */
            0xff,  /* divide  */     0xa4,  /* o with line */
            0xac,  /* u grave */     0xad,  /* u acute */
            0xae,  /* u circumflex */ 0xaf,  /* u umlaut */
            0xb4,  /* y acute */     0xb8,  /* thorn */
            0xb5   /* y umlaut */
        };
```

```
                    __LC_INDEX_END(lccollate_dummy)
                    void const *_get_lc_collate(void const *null, char const *name) {
                        return _findlocale(&lccoll_c_index, name);
                    }
                    void test_lc_collate(void) {
                        char buf[5];
                        /* test both strxfrm and strcoll here*/
                        EQS(setlocale(LC_COLLATE, NULL), "C");              /* verify starting point */
                        EQS((strxfrm(buf, "\xEF", 4), buf), "\xEF");
                        EQI(strcoll("\xEF", "j") < 0, 0);
                        EQI(!setlocale(LC_COLLATE, "ISO8859-1"), 0);   /* setlocale should work */
                        EQS(setlocale(LC_COLLATE, NULL), "ISO8859-1");
                        EQS((strxfrm(buf, "\xEF", 4), buf), "\x97");
                        EQI(strcoll("\xEF", "j") < 0, 1);
                        EQI(!setlocale(LC_COLLATE, "C"), 0);               /* setlocale should work */
                        EQS(setlocale(LC_COLLATE, NULL), "C");
                        EQS((strxfrm(buf, "\xEF", 4), buf), "\xEF");
                        EQI(strcoll("\xEF", "j") < 0, 0);
                    }
```

The `__LC_COLLATE_TRIVIAL_DEF` macro defines an array that has the element value equal to its index number. `__LC_COLLATE_TRIVIAL_DEF(lccoll_c, "C")` is equivalent to the code in Example 5-5.

Example 5-5 LC_COLLATE_DEF

```
__LC_COLLATE_DEF(lccoll_c, "C")
{
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
...
0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff
};
```

5.7.6 _get_lc_monetary()

_get_lc_monetary() must return a pointer to an __lc_monetary_blk structure. Use the macros in Example 5-6 to create the structure.

Example 5-6 LC_MONETARY_DEF

```
__LC_MONETARY_DEF(lcmonetary_c, "C",
                  "","","","","","","","",
                  255,255,255,255,255,255,255,255)
__LC_MONETARY_DEF(lcmonetary_iso8859_1, "ISO8859-1",
                  "STG ", "\243", ".", ",", "\3", "", "-",
                  2, 2, 1, 0, 1, 0, 1, 2)
__LC_INDEX_END(lcmonetary_dummy)
void const *_get_lc_monetary(void const *nullpara, char const *name) {
    return _findlocale(&lcmonetary_c_index, name);
}
void test_lc_monetary(void) {
    struct lconv lc;
    /*Test changing currency string as we change locales.*/
    EQS(setlocale(LC_MONETARY, NULL), "C");          /* verify starting point */
    _get_lconv(&lc); EQS(lc.currency_symbol, "");
    EQI(!setlocale(LC_MONETARY, "ISO8859-1"), 0);    /* setlocale should work */
    EQS(setlocale(LC_MONETARY, NULL), "ISO8859-1");
    _get_lconv(&lc); EQS(lc.currency_symbol, "\243");
    EQI(!setlocale(LC_MONETARY, "C"), 0);            /* setlocale should work */
    EQS(setlocale(LC_MONETARY, NULL), "C");   _get_lconv(&lc);
    EQS(lc.currency_symbol, "");
}
```

5.7.7   _get_lc_numeric()

_get_lc_numeric() must return a pointer to an __lc_numeric_blk structure. Use the macros in Example 5-7 to create the structure.

Example 5-7 LC_NUMERIC_DEF

```
__LC_NUMERIC_DEF(lcnumeric_c, "C",".","","")
__LC_NUMERIC_DEF(lcnumeric_iso8859_1, "ISO8859-1",
                ".", ",", "\3")
__LC_NUMERIC_DEF(lcnumeric_fr, "fr", ",", ".", "\3")
__LC_INDEX_END(lcnumeric_dummy)
void const *_get_lc_numeric(void const *null, char const *name) {
    return _findlocale(&lcnumeric_c_index, name);
}
void test_lc_numeric(void) {
    double pi = 4*atan(1.);
    char buf[20];
    /* Test changing decimal point as we shift in and out of French
     * numeric locale. */
    EQS(setlocale(LC_NUMERIC, NULL), "C");        /* verify starting point */
    snprintf(buf, sizeof(buf), "%g", pi); EQS(buf, "3.14159");
    EQI(!setlocale(LC_NUMERIC, "ISO8859-1"), 0);  /* setlocale should work */
    EQS(setlocale(LC_NUMERIC, NULL), "ISO8859-1");
    snprintf(buf, sizeof(buf), "%g", pi); EQS(buf, "3.14159");
    EQI(!setlocale(LC_NUMERIC, "fr"), 0);         /* setlocale should work */
    EQS(setlocale(LC_NUMERIC, NULL), "fr");
    snprintf(buf, sizeof(buf), "%g", pi); EQS(buf, "3,14159");
    EQI(!setlocale(LC_NUMERIC, "C"), 0);          /* setlocale should work */
    EQS(setlocale(LC_NUMERIC, NULL), "C");
    snprintf(buf, sizeof(buf), "%g", pi); EQS(buf, "3.14159");
}
```

The offset fields are interpreted similarly to __lc_monetary_blk.

5.7.8   _get_lc_time()

_get_lc_time() must return a pointer to a __lc_time_blk structure. Use the macros in
Example 5-8 to create the structure.

Example 5-8 LC_TIME_DEF

```
__LC_TIME_DEF(lctime_c, "C",
              "Sun\0Mon\0Tue\0Wed\0Thu\0Fri\0Sat",
              "Sunday\0xxx" "Monday\0xxx" "Tuesday\0xx" "Wednesday\0"
              "Thursday\0x" "Friday\0xxx" "Saturday\0",
              "Jan\0Feb\0Mar\0Apr\0May\0Jun\0Jul\0Aug\0Sep\0Oct\0Nov\0Dec",
              "January\0xx" "February\0x" "March\0xxxx" "April\0xxxx"
              "May\0xxxxxx" "June\0xxxxx" "July\0xxxxx" "August\0xxx"
              "September\0" "October\0xx" "November\0x" "December\0",
              "AM", "PM",
              "%x %X", "%d %b %Y", "%H:%M:%S")
__LC_TIME_DEF(lctime_fr, "fr",
              "dim\0lun\0mar\0mer\0jeu\0ven\0sam",
              "dimanche\0" "lundi\0xxx" "mardi\0xxx" "mercredi\0"
              "jeudi\0xxx" "vendredi\0" "samedi\0x",
              "jan\0xfev\0xmars\0avr\0xmai\0xjuin\0"
              "juil\0aout\0sep\0xoct\0xnov\0xdec\0",
              "janvier\0xx" "fevrier\0xx" "mars\0xxxxx" "avril\0xxxx"
              "mai\0xxxxxx" "juin\0xxxxx" "juillet\0xx" "aout\0xxxxx"
              "septembre\0" "octobre\0xx" "novembre\0x" "decembre\0",
              "AM", "PM", "%A, %d %B %Y, %X", "%d.%m.%y", "%H:%M:%S")
__LC_INDEX_END(lctime_dummy)
void const *_get_lc_time(void const *null, char const *name) {
    return _findlocale(&lctime_c_index, name);
}
void test_lc_time(void) {
    struct tm tm;
    char timestr[256];
    tm.tm_sec = 13;
    tm.tm_min = 13;
    tm.tm_hour = 23;
    tm.tm_mday = 12;
    tm.tm_mon = 1;
    tm.tm_year = 98;
    tm.tm_wday = 4;
    tm.tm_yday = 42;
    tm.tm_isdst = 0;
    EQS(setlocale(LC_TIME, NULL), "C");      /* verify starting point */
    strftime(timestr, sizeof(timestr), "%c", &tm);
    EQS(timestr, "12 Feb 1998 23:13:13");
    EQI(!setlocale(LC_TIME, "fr"), 0);       /* setlocale should work */
```

```
                EQS(setlocale(LC_TIME, NULL), "fr");
                strftime(timestr, sizeof(timestr), "%c", &tm);
                EQS(timestr, "jeudi, 12 fevrier 1998, 23:13:13");
                EQI(!setlocale(LC_TIME, "C"), 0);      /* setlocale should work */
                EQS(setlocale(LC_TIME, NULL), "C");
                strftime(timestr, sizeof(timestr), "%c", &tm);
                EQS(timestr, "12 Feb 1998 23:13:13");
        }
```

The offset fields are interpreted similarly to `__lc_monetary_blk`.

5.7.9   _get_lconv()

`_get_lconv()` sets the components of an `lconv` structure with values appropriate for the formatting of numeric quantities.

Syntax

void _get_lconv(**struct** lconv *lc*)

Implementation

This extension to ISO does not use any static data. If you are building an application that must conform strictly to the ISO C standard, use `localeconv()` instead.

Returns

The existing `lconv` structure *lc* is filled with formatting data.

5.7.10   localeconv()

localeconv() creates and sets the components of an lconv structure with values appropriate for the formatting of numeric quantities according to the rules of the current locale.

Syntax

struct lconv *localeconv(**void**)

Implementation

The members of the structure with type **char** * are strings. Any of these, except for decimal_point, can point to "" to indicate that the value is not available in the current locale or is of zero length.

The members with type **char** are non-negative numbers. Any of the members can be CHAR_MAX to indicate that the value is not available in the current locale.

The members included in lconv are described in *The lconv structure* on page 5-68.

Returns

The function returns a pointer to the filled-in object. The structure pointed to by the return value is not modified by the program, but might be overwritten by a subsequent call to the localeconv() function. In addition, calls to the setlocale() function with categories LC_ALL, LC_MONETARY, or LC_NUMERIC might overwrite the contents of the structure.

5.7.11   setlocale()

Selects the appropriate locale as specified by the *category* and *locale* arguments.

Syntax

`char* setlocale(int category, const char* locale)`

Implementation

The `setlocale()` function is used to change or query part or all of the current locale. The effect of the category argument for each value is described in *Locale categories*. A value of "C" for *locale* specifies the minimal environment for C translation. An empty string, "", for *locale* specifies the implementation-defined native environment. At program startup, the equivalent of `setlocale(LC_ALL, "C")` is executed.

Locale categories

The values of `category` are:

LC_COLLATE   Affects the behavior of `strcoll()`.

LC_CTYPE   Affects the behavior of the character handling functions.

LC_MONETARY   Affects the monetary formatting information returned by `localeconv()`.

LC_NUMERIC   Affects the decimal-point character for the formatted input/output functions and the string conversion functions and the numeric formatting information returned by `localeconv()`.

LC_TIME   Can affect the behavior of `strftime()`. For currently supported locales, the option has no effect.

LC_ALL   Affects all locale categories. This is the bitwise OR of all the locale categories.

Returns

If a pointer to string is given for `locale` and the selection is valid, the string associated with the specified category for the new locale is returned. If the selection cannot be honored, a null pointer is returned and the locale is not changed.

A null pointer for `locale` causes the string associated with the category for the current locale to be returned and the locale is not changed.

If *category* is LC_ALL and the most recent successful locale-setting call uses a category other than LC_ALL, a composite string might be returned. The string returned when used in a subsequent call with its associated category restores that part the program locale. The string returned is not modified by the program, but might be overwritten by a subsequent call to setlocale().

5.7.12   _findlocale()

_findlocale() searches the locale database and returns a pointer to the data block for the requested category and locale.

Syntax

void const∗ _findlocale(**void const**∗ *index*, **char const** ∗*name*)

Returns

Returns a pointer to the requested data block.

5.7.13   __LC_CTYPE_DEF

This macro is used to create CTYPE blocks. The definition from rt_locale.h and sample code are shown in Example 5-9. This example is incomplete, so refer to rt_locale.h for details.

Example 5-9 LC_CTYPE_DEF

```
#define __LC_CTYPE_DEF(sym,ln) \
static const int sym##_index = ~3 & (3 + (268+(~3 & (3 + sizeof(ln))))); \
static const char sym##_lname[~3 & (3 + sizeof(ln))] = ln; \
static const int sym##_pname = -4-(~3 & (3 + sizeof(ln))); \
static const char sym##_start = 0; \
static const char sym##_table[256] =
```

For all the macros, the first two arguments are a symbol prefix and a locale name. The resulting locale block is addressed by the expression &symprefix_start, and the index entry by the expression &symprefix_index.

Usage

See *_get_lc_ctype()* on page 5-51.

―――― **Note** ――――

Because the compiler optimizes the data segment, it reorders and removes parts of locale definitions, and breaks the data structures. The code examples provided are for informational purposes only. In practice, the definitions require additional pragmas to disable optimizations. These are specified in rt_locale.h as follows:

```
#define __lblk_start _Pragma("push_once Ono_remove_unused_constdata \
Ono_data_reorder")
#define __lblk_end _Pragma("pop")
```

5.7.14 __LC_COLLATE_DEF

This macro is used to create collate blocks used when sorting ASCII characters. The definition from rt_locale.h, the definition of a macro for creating an empty table, and sample code are shown in Example 5-10 and Example 5-11. These examples are incomplete, so refer to rt_locale.h for details.

For all the macros, the first two arguments are a symbol prefix and a locale name. The resulting locale block is addressed by the expression &symprefix_start, and the index entry by the expression &symprefix_index.

Example 5-10 Macro for use with array

```
#define __LC_COLLATE_DEF(sym,ln) \
static const int sym##_index = ~3&(3+(268+(~3&(3+sizeof(ln))))); \
static const char sym##_lname[~3 & (3 + sizeof(ln))] = ln; \
static const int sym##_pname = -4-(~3 & (3 + sizeof(ln))); \
static const int sym##_start = 4; \
static const char sym##_table[] =
```

Example 5-11 Macro that generates default table

```
#define __LC_COLLATE_TRIVIAL_DEF(sym,ln) \
static const int sym##_index = ~3&(3+(12+(~3&(3+sizeof(ln))))); \
static const char sym##_lname[~3 & (3 + sizeof(ln))] = ln; \
static const int sym##_pname = -4-(~3 & (3 + sizeof(ln))); \
static const int sym##_start = 0;
```

Usage

See *_get_lc_collate()* on page 5-53. See also *__LC_CTYPE_DEF* on page 5-63 for details of the side-effects of compiler optimizations.

5.7.15 __LC_TIME_DEF

This macro is used to create blocks used when formatting time or date values. The definition from rt_locale.h and sample code are shown in Example 5-12.

For all the macros, the first two arguments are a symbol prefix and a locale name. The resulting locale block is addressed by the expression &symprefix_start, and the index entry by the expression &symprefix_index.

Example 5-12 LC_TIME_DEF

```
#define __LC_TIME_DEF(sym,ln,wa,wf,ma,mf,am,pm,dt,df,tf) \
static const int sym##_index = ~3 & (3 + (sizeof(wa)+sizeof(wf)+sizeof(ma)+ \
sizeof(mf)+sizeof(am)+sizeof(pm)+ \
sizeof(dt)+sizeof(df)+sizeof(tf)+ \
60+(~3 & (3 + sizeof(ln)))))); \
static const char sym##_lname[~3 & (3 + sizeof(ln))] = ln; \
static const int sym##_pname = -4-(~3 & (3 + sizeof(ln))); \
static const int sym##_start = 52; \
static const int sym##_wfoff = (sizeof(wa)+52); \
static const int sym##_maoff = (sizeof(wa)+sizeof(wf)+52); \
static const int sym##_mfoff = (sizeof(wa)+sizeof(wf)+sizeof(ma)+52); \
static const int sym##_amoff = (sizeof(wa)+sizeof(wf)+sizeof(ma)+ \
sizeof(mf)+52); \
static const int sym##_pmoff = (sizeof(wa)+sizeof(wf)+sizeof(ma)+ \
sizeof(mf)+sizeof(am)+52); \
static const int sym##_dtoff = (sizeof(wa)+sizeof(wf)+sizeof(ma)+ \
sizeof(mf)+sizeof(am)+sizeof(pm)+52); \
static const int sym##_dfoff = (sizeof(wa)+sizeof(wf)+sizeof(ma)+ \
sizeof(mf)+sizeof(am)+sizeof(pm)+ \
sizeof(dt)+52); \
static const int sym##_tfoff = (sizeof(wa)+sizeof(wf)+sizeof(ma)+ \
sizeof(mf)+sizeof(am)+sizeof(pm)+ \
sizeof(dt)+sizeof(df)+52); \static const int sym##_wasiz = (sizeof(wa)/7); \
static const int sym##_wfsiz = (sizeof(wf)/7); \
static const int sym##_masiz = (sizeof(ma)/12); \
static const int sym##_mfsiz = (sizeof(mf)/12); \
static const char sym##_watxt[] = wa; \
static const char sym##_wftxt[] = wf; \
static const char sym##_matxt[] = ma; \
static const char sym##_mftxt[] = mf; \
```

```
static const char sym##_amtxt[] = am; \
static const char sym##_pmtxt[] = pm; \
static const char sym##_dttxt[] = dt; \
static const char sym##_dftxt[] = df; \
static const char sym##_tftxt[] = tf;
```

Usage

See *_get_lc_time()* on page 5-59. See also *__LC_CTYPE_DEF* on page 5-63 for details of the side-effects of compiler optimizations.

5.7.16 __LC_NUMERIC_DEF

This macro is used to create blocks used when formatting numbers. The definition from rt_locale.h and sample code are shown in Example 5-13.

For all the macros, the first two arguments are a symbol prefix and a locale name. The resulting locale block is addressed by the expression &symprefix_start, and the index entry by the expression &symprefix_index.

Example 5-13 LC_NUMERIC_DEF

```
#define __LC_NUMERIC_DEF(sym,ln,dp,ts,gr) \
static const int sym##_index = ~3 & (3 + (sizeof(dp)+sizeof(ts)+sizeof(gr)+ \
20) + (~3 & (3 + sizeof(ln)))); \
static const char sym##_lname[~3 & (3 + sizeof(ln))] = ln; \
static const int sym##_pname = -4-(~3 & (3 + sizeof(ln))); \
static const int sym##_start = 12; \
static const int sym##_tsoff = (sizeof(dp)+12); \
static const int sym##_groff = (sizeof(dp)+sizeof(ts)+12); \
static const char sym##_dptxt[] = dp; \
static const char sym##_tstxt[] = ts; \
static const char sym##_grtxt[] = gr;
```

Usage

See *_get_lc_numeric()* on page 5-58. See also *__LC_CTYPE_DEF* on page 5-63 for details of the side-effects of compiler optimizations.

5.7.17 __LC_MONETARY_DEF

This macro is used to create blocks used when formatting monetary values. The definition from `rt_locale.h` and sample code are shown in Example 5-14.

For all the macros, the first two arguments are a symbol prefix and a locale name. The resulting locale block is addressed by the expression &symprefix_start, and the index entry by the expression &symprefix_index.

Example 5-14 LC_MONETARY_DEF

```
#define __LC_MONETARY_DEF(sym,ln,ic,cs,md,mt,mg,ps,ns, \
                          id,fd,pc,pS,nc,nS,pp,np) \
static const int sym##_index = ~3 & (3 + (sizeof(ic)+sizeof(cs)+sizeof(md)+ \
                                      sizeof(mt)+sizeof(mg)+sizeof(ps)+ \
                                      sizeof(ns)+44) \
                                  + (~3 & (3 + sizeof(ln)))); \
static const char sym##_lname[~3 & (3 + sizeof(ln))] = ln; \
static const int sym##_pname = -4-(~3 & (3 + sizeof(ln))); \
static const char sym##_start = id; \
static const char sym##_fdchr = fd; \
static const char sym##_pcchr = pc; \
static const char sym##_pSchr = pS; \
static const char sym##_ncchr = nc; \
static const char sym##_nSchr = nS; \
static const char sym##_ppchr = pp; \
static const char sym##_npchr = np; \
static const int sym##_icoff = 36; \
static const int sym##_csoff = (sizeof(ic)+36); \
static const int sym##_mdoff = (sizeof(ic)+sizeof(cs)+36); \
static const int sym##_mtoff = (sizeof(ic)+sizeof(cs)+sizeof(md)+36); \
static const int sym##_mgoff = (sizeof(ic)+sizeof(cs)+sizeof(md)+ \
                                sizeof(mt)+36); \
static const int sym##_psoff = (sizeof(ic)+sizeof(cs)+sizeof(md)+ \
                                sizeof(mt)+sizeof(mg)+36); \
static const int sym##_nsoff = (sizeof(ic)+sizeof(cs)+sizeof(md)+ \
                                sizeof(mt)+sizeof(mg)+sizeof(ps)+36); \
static const char sym##_ictxt[] = ic; \
static const char sym##_cstxt[] = cs; \
static const char sym##_mdtxt[] = md; \
static const char sym##_mttxt[] = mt; \
static const char sym##_mgtxt[] = mg; \
static const char sym##_pstxt[] = ps; \
static const char sym##_nstxt[] = ns;
```

Usage

See *_get_lc_monetary()* on page 5-57. See also *__LC_CTYPE_DEF* on page 5-63 for details of the side-effects of compiler optimizations.

5.7.18   __LC_INDEX_END

This macro is used to declare the end of an index. symprefix is provided to ensure a unique name. The definition from rt_locale.h and sample code are shown in Example 5-15.

Example 5-15 LC_INDEX_END

```
#define __LC_INDEX_END(symprefix)  static const int symprefix##_index = 0;
```

5.7.19   The lconv structure

The lconv structure contains numeric formatting information. The structure is filled by the functions _get_lconv() and localeconv(). The setlocale() function must be called to initialize the lconv structure prior to using the structure in any other functions.

The definition of lconv from locale.h is shown in Example 5-16.

Example 5-16 lconv structure

```
struct lconv {
  char *decimal_point;
      /* The decimal point character used to format non-monetary quantities */
  char *thousands_sep;
      /* The character used to separate groups of digits to the left of the */
      /* decimal point character in formatted non-monetary quantities.      */
  char *grouping;
      /* A string whose elements indicate the size of each group of digits  */
      /* in formatted non-monetary quantities. See below for more details.  */
  char *int_curr_symbol;
      /* The international currency symbol applicable to the current locale.*/
      /* The first three characters contain the alphabetic international    */
      /* currency symbol in accordance with those specified in ISO 4217.    */
      /* Codes for the representation of Currency and Funds. The fourth      */
      /* character (immediately preceding the null character) is the        */
      /* character used to separate the international currency symbol from   */
      /* the monetary quantity.                                             */
```

```
      char *currency_symbol;
          /* The local currency symbol applicable to the current locale.     */
      char *mon_decimal_point;
          /* The decimal-point used to format monetary quantities.            */
      char *mon_thousands_sep;
          /* The separator for groups of digits to the left of the decimal-point*/
          /* in formatted monetary quantities.                                */
      char *mon_grouping;
          /* A string whose elements indicate the size of each group of digits */
          /* in formatted monetary quantities. See below for more details.    */
      char *positive_sign;
          /* The string used to indicate a non-negative-valued formatted      */
          /* monetary quantity.                                               */
      char *negative_sign;
          /* The string used to indicate a negative-valued formatted monetary */
          /* quantity.                                                        */
      char int_frac_digits;
          /* The number of fractional digits (those to the right of the       */
          /* decimal-point) to be displayed in an internationally formatted   */
          /* monetary quantities.                                             */
      char frac_digits;
          /* The number of fractional digits (those to the right of the       */
          /* decimal-point) to be displayed in a formatted monetary quantity. */
      char p_cs_precedes;
          /* Set to 1 or 0 if the currency_symbol respectively precedes or    */
          /* succeeds the value for a non-negative formatted monetary quantity. */
      char p_sep_by_space;
          /* Set to 1 or 0 if the currency_symbol respectively is or is not   */
          /* separated by a space from the value for a non-negative formatted */
          /* monetary quantity.                                               */
      char n_cs_precedes;
          /* Set to 1 or 0 if the currency_symbol respectively precedes or    */
          /* succeeds the value for a negative formatted monetary quantity.   */
      char n_sep_by_space;
          /* Set to 1 or 0 if the currency_symbol respectively is or is not   */
          /* separated by a space from the value for a negative formatted     */
          /* monetary quantity.                                               */
      char p_sign_posn;
          /* Set to a value indicating the position of the positive_sign for a */
          /* non-negative formatted monetary quantity. See below for more details*/
      char n_sign_posn;
          /* Set to a value indicating the position of the negative_sign for a */
          /* negative formatted monetary quantity. */
  };
```

The elements of `grouping` and `non_grouping` (shown in Example 5-16 on page 5-68) are interpreted as follows:

CHAR_MAX

No additional grouping is to be performed.

0   The previous element is repeated for the remainder of the digits.

other   The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

The value of `p_sign_posn` and `n_sign_posn` (shown in Example 5-16 on page 5-68) are interpreted as follows:

0   Parentheses surround the quantity and currency symbol.

1   The sign string precedes the quantity and currency symbol.

2   The sign string is after the quantity and currency symbol.

3   The sign string immediately precedes the currency symbol.

4   The sign string immediately succeeds the currency symbol.

5.8   Tailoring error signaling, error handling, and program exit

All trap or error signals raised by the C library go through the `__raise()` function. You can re-implement this function or the lower-level functions that it uses.

———— **Caution** ————

The IEEE 754 standard for floating-point processing states that the default response to an exception is to proceed without a trap. You can modify floating-point error handling by tailoring the functions and definitions in `fenv.h`. See also Chapter 6 *Floating-point Support*.

See the `rt_misc.h` include file for more information on error-related functions.

The trap and error-handling functions are shown in Table 5-11. See also *Tailoring the C library to a new execution environment* on page 5-35 for additional information about application initialization and shutdown.

Table 5-11 Trap and error handling

Function	Description
`_sys_exit()`	Called, eventually, by all exits from the library. See *_sys_exit()* on page 5-72.
`errno`	Is a static variable used with error handling. See *errno* on page 5-72.
`__rt_errno_addr()`	This function is called to obtain the address of the variable errno. See *__rt_errno_addr()* on page 5-73.
`__raise()`	Raises a signal to indicate a runtime anomaly. See *__raise()* on page 5-73.
`__rt_raise()`	Raises a signal to indicate a runtime anomaly. See *__rt_raise()* on page 5-74.
`__default_signal_handler()`	Displays an error indication to the user. See *__default_signal_handler()* on page 5-75.
`_ttywrch()`	The default implementation of _ttywrch() is semihosted and therefore it uses the semihosting SWI. See *_ttywrch()* on page 5-75.
`__rt_fp_status_addr()`	This function is called to obtain the address of the fp status word. See *__rt_fp_status_addr()* on page 5-76.

5.8.1   _sys_exit()

The library exit function. All exits from the library eventually call _sys_exit().

Syntax

void _sys_exit(**int** *return_code*)

Implementation

This function must not return. You can intercept application exit at a higher level by either:

•   Implementing the C library function exit() as part of your application. You lose atexit() processing and library shutdown if you do this.

•   Implementing the function __rt_exit(int n) as part of your application. You lose library shutdown if you do this, but atexit() processing is still performed when exit() is called or main() returns.

——— **Caution** ———

This function is called if a stack overflow occurs. If you re-implement this function and include a stack check as part of the code, the overflow causes an immediate return to _sys_exit() causing a worse stack overflow. It is not recommended that this function performs stack checking.

Returns

The return code is advisory. An implementation might attempt to pass it to the execution environment.

5.8.2   errno

The C library errno variable is defined in the implicit static data area of the library. This area is identified by __user_libspace(). It occupies part of initial stack space used by the functions that established the runtime stack. The definition of errno is:

(∗(**volatile int** ∗) __rt_errno_addr())

You can define __rt_errno_addr() if you want to place errno at a user-defined location instead of the default location identified by __user_libspace(). Also, see *The __user_libspace static data area* on page 5-8.

Returns

The default implementation is a veneer on `__user_libspace()` that returns the address of the status word. A suitable default definition is given in the C library standard headers.

5.8.3 __rt_errno_addr()

This function is called to get the address of the C library `errno` variable when the C library attempts to read or write `errno`. A default implementation is provided by the library. It is unlikely that you have to re-implement this function.

Syntax

volatile int *`__rt_errno_addr`(**void**)

5.8.4 __raise()

This function raises a signal to indicate a runtime anomaly.

Syntax

int `__raise`(**int** *major*, **int** *minor*)

where:

major   Is an integer that holds the signal number.

minor   Is an integer or string constant or variable.

Implementation

This function calls the normal C signal mechanism or the default signal handler. See also *_ttywrch()* on page 5-75 for more information. Also, see *Thread-safety in the ARM C libraries* on page 5-12.

You can replace the `__raise()` function by defining:

int `__raise`(**int** *signal*, **int** *argument*)

This enables you to bypass the C signal mechanism and its data-consuming signal handler vector, but otherwise gives essentially the same interface as:

void `__default_signal_handler`(**int** *signal*, **int** *arg*)

Returns

There are three possibilities for __raise() return condition:

no return   The handler performs a long jump or restart.

0   The signal was handled.

nonzero   The calling code must pass that return value to the exit code. The default
                 library implementation calls _sys_exit(rc) if __raise() returns a
                 nonzero return code *rc*.

5.8.5   __rt_raise()

This function raises a signal to indicate a runtime anomaly.

Syntax

void __rt_raise(**int** *signal*, **int** *type*)

where:

signal   Is an integer that holds the signal number.

type   Is an integer or string constant or variable.

Implementation

Redefine this to replace the entire signal handling mechanism for the library. The
default implementation calls __raise(). See *__raise()* on page 5-73 for more
information.

Depending on the value returned from __raise():

no return   The handler performed a long jump or restart and __rt_raise() does not
                 regain control.

0   The signal was handled and __rt_raise() exits.

nonzero   The default library implementation calls _sys_exit(rc) if __raise()
                 returns a nonzero return code *rc*.

5.8.6 __default_signal_handler()

This function handles a raised signal. The default action is to print an error message and exit.

Syntax

void __default_signal_handler(**int** *signal*, **int** *arg*)

Implementation

The default signal handler uses _ttywrch() to print a message and calls _sys_exit() to exit. You can replace the default signal handler by defining:

void __default_signal_handler(**int** *signal*, **int** *argument*)

The interface is the same as __raise(), but this function is only called after the C signal handling mechanism has declined to process the signal.

A complete list of the defined signals is in signal.h. See Table 5-19 on page 5-115 for those signals that are used by the libraries.

——— **Note** ———

The signals used by the libraries might change in future releases of the product.

5.8.7 _ttywrch()

This function writes a character to the console. The console might have been redirected. You can use this function as a last resort error handling routine.

Syntax

void _ttywrch(**int** *ch*)

Implementation

The default implementation of this function uses the semihosting SWI.

You can redefine this function, or __raise(), even if there is no other input/output. For example, it might write an error message to a log kept in nonvolatile memory.

5.8.8   __rt_fp_status_addr()

This function returns the address of the floating-point status register.

Syntax

```
unsigned* _rt_fp_status_addr(void)
```

Implementation

If `__rt_fp_status_addr()` is not defined, the default implementation from the C library is used. The value is initialized when `__rt_lib_init()` calls `_fp_init()`. The constants for the status word are listed in `fenv.h`. The default fp status is 0.

Also see *Thread-safety in the ARM C libraries* on page 5-12.

                             ARM DUI 0282B

5.9 Tailoring storage management

This section describes the functions from `rt_heap.h` that you can define if you are tailoring memory management. There are also two helper functions that you can call from your heap implementation.

See the `rt_heap.h` and `rt_memory.s` include files for more information on memory-related functions.

———— **Note** ————

If you are developing embedded systems with limited RAM you might require a system that does not use the heap or any heap-using functions. Otherwise, you might require your own heap functions. There are two library functions that you can include to cause a warning message if the heap is used:

`__use_no_heap()`

> Guards against use of `malloc()`, `realloc()`, `free()`, and any function that uses them (such as `calloc()` and `stdio`). This only guards against the use of these functions from the libraries supplied by ARM Limited, not your own libraries.

`__use_no_heap_region()`

> Has the same properties as `__use_no_heap()`, but also guards against other things that use the heap memory region. For example, if you declare `main()` as a function taking arguments, the heap region is used for collecting `argc` and `argv`.

This section describes:

5.9.1   Support for malloc

`malloc()`, `realloc()`, `calloc()`, and `free()` are built on a heap abstract data type. You can either:

- choose between Heap1 or Heap2, the two provided heap implementations

- write your own heap implementation of the abstract data type for heap (see *Creating your own storage-management system* on page 5-80).

The default implementations of `malloc()`, `realloc()`, and `calloc()` maintain an eight-byte aligned heap.

Heap1: Standard heap implementation

Heap1, the default implementation, implements the smallest and simplest heap manager. The heap is managed as a singly-linked list of free blocks held in increasing address order. The allocation policy is first-fit by address.

This implementation has low overheads, but the cost of `malloc()` or `free()` grows linearly with the number of free blocks. The smallest block that can be allocated is four bytes and there is an additional overhead of four bytes. If you expect more than 100 unallocated blocks it is recommended that you use Heap2.

Heap2: Alternative heap implementation

Heap2 provides a compact implementation with the cost of `malloc()` or `free()` growing logarithmically with the number of free blocks. The allocation policy is first-fit by address. The smallest block that can be allocated is 12 bytes and there is an additional overhead of four bytes.

Heap2 is recommended when you require near constant-time performance in the presence of hundreds of free blocks. To select the alternative standard implementation, use either:

- `IMPORT __use_realtime_heap` from assembly language
- `#pragma import(__use_realtime_heap)` from C.

You can also define your own heap implementation. See *Creating your own storage-management system* on page 5-80 for more information.

Using Heap2

The Heap2 real-time heap implementation must know the maximum address space the heap spans. The smaller the address range, the more efficient the algorithm is.

By default, the heap extent is taken to be 16MB starting at the beginning of the heap (defined as the start of the first chunk of memory given to the heap manager by `__rt_initial_stackheap()` or `__rt_heap_extend()`).

The heap bounds are given by:

```
struct __heap_extent {
      unsigned base, range;};
__value_in_regs struct __heap_extent __user_heap_extent(
      unsigned defaultbase, unsigned defaultsize);
```

The function prototype for `__user_heap_extent()` is in `rt_misc.h`.

The Heap1 algorithm does not require the bounds on the heap extent, therefore it never calls this function.

You must redefine `__user_heap_extent()` if:

*   you require a heap to span more than 16MB of address space

*   your memory model can supply a block of memory at a lower address than the first one supplied.

If you know in advance that the address space bounds of your heap are small, you do not have to redefine `__user_heap_extent()`, but it does speed up the heap algorithms if you do.

The input parameters are the default values that are used if this routine is not defined. You can, for example, leave the default base value unchanged and only adjust the size.

———— **Note** ————

The size field returned must be a power of two. If you return zero for `size`, the heap extent is set to 4GB.

Using a heap implementation from bare machine C

To use a heap implementation in an application that does not define `main()` and does not initialize the C library:

1.   Call `_init_alloc(`*base*`, `*top*`)` to define the base and top of the memory you want to manage as a heap.

2.   Define the function `unsigned __rt_heap_extend(`**unsigned** `size, `**void** `**block)` to handle calls to extend the heap when it becomes full.

alloca()

alloca() behaves identically to malloc() except that alloca() has automatic garbage collection (see *alloca()* on page 5-124).

5.9.2   Creating your own storage-management system

You can implement the heap functions in Table 5-12 to create a new storage-management system.

Table 5-12 Heap functions

Function	Description
__Heap_Descriptor	You must define your own implementation of the abstract data type for heap. See *__Heap_Descriptor* on page 5-81.
__Heap_Initialize()	Initializes the heap. See *__Heap_Initialize()* on page 5-81.
__Heap_DescSize()	Returns the size of the __Heap_Descriptor structure. See *__Heap_DescSize()* on page 5-82.
__Heap_ProvideMemory()	Called to increase the size of the heap. See *__Heap_ProvideMemory()* on page 5-82.
__Heap_Alloc()	Allocates memory from the heap to the application. See *__Heap_Alloc()* on page 5-83.
__Heap_Free()	Returns previously allocated space to the heap. See *__Heap_Free()* on page 5-83.
__Heap_Realloc()	Adjusts the size of an already allocated block. See *__Heap_Realloc()* on page 5-84.
__Heap_Stats()	Called from __heapstats() to print statistics about the state of the heap. See *__Heap_Stats()* on page 5-84.
__Heap_Valid()	Called to perform a consistency check on the heap. See *__Heap_Valid()* on page 5-85.
__Heap_Full()	Attempts to acquire a new block from the system. You must not re-implement this function. See *__Heap_Full()* on page 5-85.
__Heap_Broken()	Called when an inconsistency in the heap is detected. See *__Heap_Broken()* on page 5-86.

5.9.3 __Heap_Descriptor

You must define your own implementation of the abstract data type for heap. A C header file describing this abstract data type is provided in rt_heap.h. You must provide the interior definition of the structure so that the other functions can find the heap data. Typical contents are given in Example 5-17.

Example 5-17 Heap_Descriptor

```
struct __Heap_Descriptor {
  void *my_first_free_block;
  void *my_heap_limit;
}
```

Your heap descriptor is set by __Heap_Initialize() and is passed to the other heap functions, for example __Heap_Alloc() and __Heap_Free().

5.9.4 __Heap_Initialize()

Initializes the heap.

Syntax

void __Heap_Initialize(**struct** __Heap_Descriptor *h)

Implementation

This is called at initialization. You must redefine it to set up the fields in your heap descriptor structure to correct initial values. A typical linked-list heap initializes the *first_free_block* pointer to NULL to indicate that there are no free blocks in the heap.

5.9.5   __Heap_DescSize()

Returns the size of the __Heap_Descriptor structure.

Syntax

int __Heap_DescSize(**int** 0)

Implementation

This is called at initialization. It must return the size of your heap descriptor structure. In almost all cases the implementation in Example 5-18 is sufficient.

Example 5-18 Heap_DescSize

```
extern int __Heap_DescSize(int zero) {return sizeof(__Heap_Descriptor);}
```

This routine is required so that the library initialization can find an initial piece of memory big enough to be the heap descriptor.

5.9.6   __Heap_ProvideMemory()

Called to increase the size of the heap.

Syntax

void __Heap_ProvideMemory(**struct** __Heap_Descriptor *h*, **void** *base*, size_t *size*)

Implementation

This is called when the system provides a chunk of memory for use by the heap. The parameters are:

*   your heap descriptor
*   a pointer to a new eight-byte aligned block of memory
*   the size of the block.

__Heap_ProvideMemory() can assume that the input block is eight-byte aligned. A typical __Heap_ProvideMemory() implementation might set up the new block of memory as a free-list entry and add it to the free chain.

5.9.7   __Heap_Alloc()

Allocates memory from the heap to the application.

Syntax

void *__Heap_Alloc(**struct** __Heap_Descriptor *h, size_t *size*)

Implementation

This is called from malloc(), and must return a pointer to *size* bytes of memory allocated from the heap, or NULL if nothing can be allocated. You must ensure that the size of the block can be determined when it is time to free it. Typically, the returned block size is stored in the word immediately before its start address. The default implementation of this function allocates an eight-byte aligned block of memory. If you re-implement this function it is recommended that you return eight-byte aligned blocks of memory.

5.9.8   __Heap_Free()

Returns previously allocated space to the heap.

Syntax

void __Heap_Free(**struct** __Heap_Descriptor *h, **void** *_blk*)

Implementation

This is called from free(), and given a pointer that was previously returned from either __Heap_Alloc() or __Heap_Realloc(). It returns the previously allocated space to the collection of free blocks in the heap.

5.9.9 __Heap_Realloc()

Adjusts the size of an already allocated block.

Syntax

void *__Heap_Realloc(**struct** __Heap_Descriptor *h, **void** *_blk, size_t *size*)

Implementation

This is called from realloc(). It is never passed trivial cases such as *blk* equal to NULL or *size* equal to zero. It adjusts the size of the allocated block *blk* to become *size*. The reallocation might involve moving the block, copying as much of the data as is common to the old and new sizes, and returning the new address. The default implementation of this function maintains eight-byte alignment of the heap block. If you re-implement this function it is recommended that you maintain eight-byte alignment.

5.9.10 __Heap_Stats()

Called from __heapstats() to print statistics about the state of the heap.

Syntax

void *__Heap_Stats(__Heap_Descriptor *h, **int**(*print)(**void** *, **char const** *format*,...), **void** *printparam*)

Implementation

It must output its results, using the supplied printf-type print routine, by calls of the form:

```
print(printparam, "%d free blocks\n", nblocks);
```

The format of the statistics data is implementation-defined, so it can do nothing. This routine is effectively optional, because it is never called unless the user program calls __heapstats().

5.9.11 __Heap_Valid()

Called from __heapvalid() to perform a consistency check on the heap data structures and attempt to identify an invalid or corrupted heap.

Syntax

int __Heap_Valid(**struct** __Heap_Descriptor *h, **int**(*print)(**void** *, **char const**
*format,...), **void** *printparam, **int** verbose)

Implementation

It must output error messages and diagnostics using the supplied printf-type print routine. For example, by a call of the form:

print(printparam, "free block at %p is corrupt\n",block_addr);

This routine is effectively optional, because it is never called unless the user program calls __heapvalid().

Returns

The function must return nonzero if the heap is valid or zero if the heap is corrupted. It must use print to output error messages if it finds problems in the heap. If the *verbose* parameter is nonzero, it can also output diagnostic data.

5.9.12 __Heap_Full()

Attempts to acquire a new block of at least *size* bytes from the system. You must not re-implement this function.

Syntax

int __Heap_Full(**struct** __Heap_Descriptor *h, size_t *size*)

Implementation

If __Heap_Alloc() or __Heap_Realloc() cannot allocate a block of the required size from the memory owned by the heap, then before giving up and returning NULL, they can try calling this routine.

You must provide space for heap housekeeping data. If the user asks for 1000 bytes and you store a word before every allocated block, you must ask __Heap_Full() for 1004 bytes, not 1000.

Before calling `__Heap_Full()`, you must ensure that the heap data structures are in a consistent state so that `__Heap_ProvideMemory()` calls can add the new block to the heap successfully.

Returns

If `__Heap_Full()` is successful, it calls `__Heap_ProvideMemory()` to add the new block to the heap, and return nonzero. If it fails, it returns 0.

5.9.13   __Heap_Broken()

Called when an inconsistency in the heap is detected. You must not re-implement this function.

Syntax

`int __Heap_Broken(struct __Heap_Descriptor *h)`

Implementation

If `__Heap_Alloc()`, `__Heap_Realloc()`, `__Heap_Free()`, or `__Heap_ProvideMemory()` detect an inconsistency in the heap structures they can call this function to terminate the program with a suitable error message.

5.10 Tailoring the runtime memory model

This section describes:

- the management of writable memory by the C library as static data, heap, and stack

- functions that can be redefined to change how writable memory is managed.

This section includes:
- *The memory models*
- *Controlling the runtime memory model* on page 5-88
- *Writing your own memory model* on page 5-88
- *__user_initial_stackheap()* on page 5-89
- *__user_heap_extend()* on page 5-90
- *__user_heap_extent()* on page 5-91
- *__user_stack_slop()* on page 5-91
- *__rt_stackheap_init()* on page 5-92
- *__rt_stack_overflow()* on page 5-92
- *__rt_heap_extend()* on page 5-93
- *__rt_stack_postlongjmp()* on page 5-94.

5.10.1 The memory models

You can select either of the following memory models:

Single memory region

The stack grows downward from the top of the memory region. The heap grows upwards from the bottom of the region. This is the default. The memory managed by the heap never shrinks. Heap memory that is freed by calling `free()` cannot be reused for other purposes.

Two memory regions

One memory region is for the stack and the other is for the heap. The size of the heap region can be zero. The stack region can be in allocated memory or inherited from the execution environment.

To use the two-region model rather than the default single-region model, use either:

- `IMPORT __use_two_region_memory` from assembly language
- `#pragma import(__use_two_region_memory)` from C.

—— **Note** ——

If you use the two-region memory model and do not provide any heap memory, you cannot call malloc(), use stdio, or get command-line arguments for main().

If you set the size of the heap region to zero and define __user_heap_extend() as a function that can extend the heap, the heap is created when it is required.

See the description of __use_no_heap() in *Tailoring storage management* on page 5-77, for details on how to issue a warning message if the heap or heap region is used.

5.10.2   Controlling the runtime memory model

The behavior of the heap and stack manager can be modified by redefining the functions listed in Table 5-13.

Table 5-13 Memory model initialization

Function	Description
__user_initial_stackheap()	Returns the location of the initial heap. See *__user_initial_stackheap()* on page 5-89.
__user_heap_extend()	Returns the size and base address of a heap extra block. See *__user_heap_extend()* on page 5-90.
__user_stack_slop()	Returns the amount of extra stack. See *__user_stack_slop()* on page 5-91.

The hidden static data for the library is provided by __user_libspace. The static data area is also used as a stack during the library initialization process. This function does not normally require re-implementation. See *Tailoring static data access* on page 5-45.

5.10.3   Writing your own memory model

If the provided memory models do not meet your requirements, you can write your own. A memory model must define the functions described in Table 5-14 on page 5-89. All functions are ARM state functions. The library takes care of entry from Thumb state if this is required. An incomplete prototype implementation for the model is provided in rt_memory.s located in the ...\include directory.

Use the prototype as a starting point for your own implementation.

Table 5-14 Memory model functions

Function	Description
__rt_stackheap_init()	Sets the application stack and initial heap. See *__rt_stackheap_init()* on page 5-92.
__rt_heap_extend()	Returns a new block of memory to add to the heap. See *__rt_heap_extend()* on page 5-93.
__rt_stack_postlongjmp()	Atomically sets the stack pointer and stack limit pointer to their correct values after a call to longjmp. See *__rt_stack_postlongjmp()* on page 5-94.
__rt_stack_overflow()	Handles stack overflows. (This is only required to be implemented for stack-checked variants.) See *__rt_stack_overflow()* on page 5-92.

5.10.4 __user_initial_stackheap()

Returns the locations of the initial stack and heap.

Syntax

__value_in_regs **struct** __initial_stackheap __user_initial_stackheap(**unsigned** *R0*, **unsigned** *SP*, **unsigned** *R2*, **unsigned** *SL*)

Implementation

——— **Note** ———

If you are using scatter-loading files with the linker, you must re-implement this function. The default implementation uses the value of the symbol Image$$ZI$$Limit. This symbol is not defined if the linker uses a scatter-loading file (specified with the --scatter command-line option).

If this function is redefined, it must:

* use no more than 88 bytes of stack
* not corrupt registers other than r12 (ip)
* return in r0-r3 respectively the heap base, stack base, heap limit, and stack limit
* maintain eight-byte alignment of the heap.

For the default single region model, the values in r2 and r3 are ignored and all memory between r0 and r1 is available for the heap. For a two region model, the heap limit is set by r2 and the stack limit is set by r3.

The values of sp (r13) and sl (r10) at the time __main() is called are passed as arguments in r1 and r3, respectively. The default implementation of __user_initial_stackheap() that uses the semihosting SWI SYS_HEAPINFO is given by the library in module sys_stackheap.o.

To create a version of __user_initial_stack_heap() that inherits sp and sl from the execution environment and does not have a heap, set r0 and r2 to the value of r3 and return.

The definition of __initial_stackheap in rt_misc.h is:

struct __initial_stackheap{ unsigned heap_base, stack_base, heap_limit, stack_limit;}

―――― **Note** ――――

The value of stack_base is 0x1 greater than the highest address used by the stack.

Refer to the examples directory for example re-implementations of this function.

Returns

The values returned in r0 to r3 depend on whether you are using the one or two region model:

One region   (r0,r1) is the single stack and heap region. r1 is greater than r0. r2 and r3 are ignored.

Two regions (r0, r2) is the initial heap and (r3, r1) is the initial stack. r2 is greater than or equal to r0. r3 is less than r1.

5.10.5   __user_heap_extend()

This function can be defined to return extra blocks of memory, separate from the initial one, to be used by the heap. If defined, this function must return the size and base address of an eight-byte aligned heap extension block.

Syntax

unsigned __user_heap_extend(**int** 0, **void** **base*, **unsigned** *requested_size*)

Implementation

There is no default implementation of this function. If you define this function, it must have the following characteristics:

- The returned size must be either:
    — a multiple of eight bytes of at least the requested size
    — 0, denoting that the request cannot be honored.

- Size is measured in bytes.

- The function is subject only to AAPCS constraints.

- The first argument is always zero on entry and can be ignored. The base is returned in the register holding this argument.

- The returned base address must be aligned on an eight-byte boundary.

5.10.6 __user_heap_extent()

If defined, this function returns the base address and maximum range of the heap.

Syntax

```
__value_in_regs struct __heap_extent __user_heap_extent(unsigned ignore1,
unsigned ignore2)
```

Implementation

There is no default implementation of this function. The values of the parameters *ignore1* and *ignore2* are not used by the function.

5.10.7 __user_stack_slop()

If defined, this function returns the size of the extra stack your system requires below sl. The extra stack is in addition to the 256 bytes required by AAPCS. The extra space might enable an interrupt handler to execute on your stack or enable a chain of unchecked functions calls.

Syntax

```
__value_in_regs struct __stack_slop __user_stack_slop(unsigned ignore,
unsigned ignore)
```

Implementation

There is no default implementation of this function.

Returns

If you define this function, it must return the following values in registers:

r0   The amount of extra stack (measured in bytes) that must always be available so an interrupt handler can execute on the stack at an arbitrary instant.

r1   The amount of extra stack (measured in bytes) that must be available after stack overflow to support recovery from overflow.

5.10.8   __rt_stackheap_init()

This function is responsible for setting up sp and sl to point at a valid stack, and must also return in r0 and r1 the lower and upper bounds of a chunk of memory that can be used as a heap. (It can decline to do the latter, by returning r0 equal to r1. In this case, the first call to malloc() results in a call to __rt_heap_extend(), described in *__rt_heap_extend()* on page 5-93.) An incomplete prototype implementation is in rt_memory.s. Because it is the first function called from entry, it does not have to preserve any other registers. On entry to this function, sp and sl are exactly as they were on entry to the whole application, so a valid stack can be inherited from the execution environment if desired. (sl is only required if stack checking is used.)

5.10.9   __rt_stack_overflow()

This function is called if a stack overflow occurs. An incomplete prototype implementation is in rt_memory.s.

Implementation

This function is called with r12 (ip) equal to the desired new sp, and with sp up to 256 bytes below sl.

If your memory model is used only with the default non stack-checked AAPCS, you do not have to implement this function.

The stack overflow routines are called at function entry if a stack limit check fails. These are subject to the usual register-use restrictions on stack overflow routines. In particular, they cannot use r0-r3 because the arguments are still held there, and they cannot use registers r4 to r11 in case the routine did not save them.

Returns

The function does not return to `lr`. It must return by branching to
`__rt_stack_overflow_return`.

5.10.10 __rt_heap_extend()

This function returns a new eight-byte aligned block of memory to add to the heap, if
possible. If you re-implement the other memory model functions, you must
re-implement this function. An incomplete prototype implementation is in `rt_memory.s`.

Implementation

The calling convention is ordinary AAPCS. On entry, `r0` is the minimum size of the
block to add, and `r1` holds a pointer to a location to store the base address.

The default implementation has the following characteristics:

- The returned size is either:
  — a multiple of eight bytes of at least the requested size
  — 0, denoting that the request cannot be honored.

- The returned base address is aligned on an eight-byte boundary.

- Size is measured in bytes.

- The function is subject only to AAPCS constraints.

Returns

The default implementation extends the heap if there is sufficient free heap memory. If
it cannot, it calls `__user_heap_extend()` if it is implemented (see *__user_heap_extend()*
on page 5-90). On exit, `r0` is the size of the block acquired, or 0 if nothing could be
obtained, and the memory location `r1` pointed to on entry contains the base address of
the block.

5.10.11 __rt_stack_postlongjmp()

This function sets sp and sl to correct values after a call to longjmp(). An incomplete prototype implementation in assembler code is in rt_memory.s.

Implementation

This function is called with r0 containing the pre-setjmp() value for sl, and r1 containing the pre-setjmp() value for sp.

If your memory model is used only with non stack-checked AAPCS, you do not have to implement this function.

Returns

The function must set sl and sp to valid post-longjmp() values. The registers must be set atomically to avoid interrupt problems. So in the minimal implementation where the memory model requires no special handling, you would push r0 and r1 on the stack and then use LDM to load sl and sp atomically with the new values.

5.11   Tailoring the input/output functions

The higher-level input/output functions such as `fscanf()` and `fprintf()` are not target-dependent. However, the higher-level functions perform input/output by calling lower-level functions that are target-dependent. To retarget input/output, you can either avoid these higher-level functions or redefine the lower-level functions.

See the `rt_sys.h` include file for more information on I/O functions.

Also see *Writing reentrant and thread-safe code* on page 5-6.

This section includes:
* *Dependencies on low-level functions*
* *Target-dependent input/output support functions* on page 5-99
* *_sys_open()* on page 5-100
* *_sys_close()* on page 5-100
* *_sys_read()* on page 5-101
* *_sys_write()* on page 5-102
* *_sys_ensure()* on page 5-103
* *_sys_flen()* on page 5-103
* *_sys_seek()* on page 5-104
* *_sys_istty()* on page 5-104
* *_sys_tmpnam()* on page 5-105
* *_sys_command_string()* on page 5-105.

5.11.1   Dependencies on low-level functions

The dependencies of the higher-level function on lower-level functions is shown in Table 5-15. If you define your own versions of the lower-level functions, you can use the library versions of the higher-level functions directly. `fgetc()` uses `__FILE`, but `fputc()` uses `__FILE` and `ferror()`.

Table 5-15 Input/output dependencies

Low-level object	Description	fprintf	printf	fwrite	fputs	puts	fscanf	scanf	fread	read	fgets	gets
`__FILE`	The file structure	x	x	x	x	x	x	x	x	x	x	x
`__stdin`	The standard input object of type `__FILE`	-	-	-	-	-	-	x	-	x	-	x
`__stdout`	The standard output object of type `__FILE`	-	x	-	-	x	-	-	-	-	-	-
`fputc()`	Outputs a character to a file	x	x	x	x	x	-	-	-	-	-	-

Table 5-15 Input/output dependencies

Low-level object	Description	fprintf	printf	fwrite	fputs	puts	fscanf	scanf	fread	read	fgets	gets
ferror()	Returns the error status accumulated during file input/output	x	x	x	-	-	-	-	-	-	-	-
fgetc()	Gets a character from a file	-	-	-	-	-	x	x	x	x	x	x
__backspace()	Moves file pointer to previous character. See *Re-implementing __backspace()* on page 5-98.	-	-	-	-	-	x	x	-	-	-	-

See the ISO C Reference for syntax of the low-level functions.

———— **Note** ————

If you choose to re-implement fgetc(), fputc(), and __backspace(), be aware that fopen() and related functions use the ARM layout for the FILE structure.

printf family

The printf family consists of _printf(), printf(), _fprintf(), fprintf(), vprintf(), and vfprintf(). All these functions use __FILE opaquely and depend only on the functions fputc() and ferror(). The functions _printf() and _fprintf() are identical to printf() and fprintf() except that they cannot format floating-point values.

The standard output functions of the form _printf(...) are equivalent to:

fprintf(& __stdout, ...)

where __stdout has type __FILE.

scanf family

The scanf() family consists of scanf() and fscanf(). These functions depend only on the functions fgetc(), __FILE, and __backspace(). See *Re-implementing __backspace()* on page 5-98.

The standard input function of the form scanf(...) is equivalent to:

fscanf(& __stdin, ...)

where __stdin has type __FILE.

fwrite(), fputs, and puts

If you define your own version of __FILE, and your own fputc() and ferror() functions and the __stdout object, you can use all of the printf() family, fwrite(), fputs(), and puts() unchanged from the library. Example 5-19 shows how to do this. Consider modifying the system routines if you require real file handling.

Example 5-19 printf() and __FILE

```
#include <stdio.h>
struct __FILE {
    int handle;
    /* Whatever you need here (if the only files you are using
        is the stdoutput using printf for debugging, no file
        handling is required) */
};
FILE __stdout;
int fputc(int ch, FILE *f)
{
    /* Your implementation of fputc */
    return ch;
}
int ferror(FILE *f)
{
    /* Your implementation of ferror */
    return EOF;
}
void test(void)
{
    printf("Hello world\n");  /* This works ... */
}
```

By default, fread() and fwrite() call fast block input/output functions that are part of the ARM stream implementation. If you define your own __FILE structure instead of using the ARM stream implementation, fread() and fwrite() call fgetc() instead of calling the block input/output functions. See also the implementation in the main examples directory, in ...\emb_sw_dev\source\retarget.c.

fread(), fgets(), and gets()

The functions fread(), fgets(), and gets() are implemented as a loop over fgetc() and ferror(). Each uses the FILE argument opaquely.

If you provide your own implementation of __FILE, __stdin (for gets()), fgetc(), and ferror(), you can use these functions directly from the library.

Re-implementing __backspace()

The function __backspace() is used by the scanf family of functions. It must never be called directly, but re-implemented if you are retargeting the stdio arrangements at the fgetc() level.

The semantics are:

int __backspace(FILE *stream);

__backspace(stream) must be called after reading a character from the stream. It returns to the stream the last character that was read from the stream, so that the same character is read from the stream again.

__backspace is separate from ungetc(). This is to guarantee that a single character can be pushed back after the scanf family of functions has finished.

The value returned by __backspace() is either 0 (success) or EOF (failure). It returns EOF only if used incorrectly, for example, if no characters have been read from the stream. When used correctly, __backspace() must always return 0, because the scanf family of functions do not check the error return.

The interaction between __backspace() and ungetc() is:

*   If you apply __backspace() to a stream and then ungetc() a character into the same stream, subsequent calls to fgetc() must return first the character returned by ungetc(), and then the one returned by __backspace().

*   If you ungetc() a character back to a stream, then read it with fgetc(), and then backspace it, the next character read by fgetc() must be the same character that was returned to the stream. That is the __backspace() operation must cancel the effect of the fgetc() operation. However, another call to ungetc() after the call to __backspace() is not required to succeed.

*   The situation where you ungetc() a character into a stream and then __backspace() another one immediately, with no intervening read, never arises. __backspace() must only be called after fgetc(), so this sequence of calls is illegal. You can write __backspace() implementations assuming that this will not happen.

5.11.2 Target-dependent input/output support functions

rt_sys.h defines the type FILEHANDLE. The value of FILEHANDLE is returned by _sys_open() and identifies an open file on the host system.

The target-dependent input and output functions and their library members are listed in Table 5-16.

Table 5-16 I/O support functions

Function	Description
_sys_open()	*_sys_open()* on page 5-100
_sys_close()	*_sys_close()* on page 5-100
_sys_read()	*_sys_read()* on page 5-101
_sys_write()	*_sys_write()* on page 5-102
_sys_seek()	*_sys_seek()* on page 5-104
_sys_ensure()	*_sys_ensure()* on page 5-103
_sys_flen()	*_sys_flen()* on page 5-103
_sys_istty()	*_sys_istty()* on page 5-104
_sys_tmpnam()	*_sys_tmpnam()* on page 5-105
_sys_command_string()	*_sys_command_string()* on page 5-105

The default implementation of these functions is semihosted. That is, each function uses the semihosting SWI. If any function is redefined, all stream-support functions must be redefined.

5.11.3 **_sys_open()**

This function opens a file.

Syntax

FILEHANDLE _sys_open(**const char** *name, **int** openmode)

Implementation

The _sys_open function is required by fopen() and freopen(). These functions, in turn, are required if any file input/output function is to be used.

The *openmode* parameter is a bitmap, whose bits mostly correspond directly to the ISO mode specification. See rt_sys.h for details. Target-dependent extensions are possible, but freopen() must also be extended.

Returns

The return value is –1 if an error occurs.

5.11.4 **_sys_close()**

This function closes a file previously opened with _sys_open().

Syntax

int _sys_close(FILEHANDLE fh)

Implementation

This function must be defined if any input/output function is to be used.

Returns

The return value is 0 if successful. A nonzero value indicates an error.

5.11.5   _sys_read()

This function reads the contents of a file into a buffer.

Syntax

int _sys_read(FILEHANDLE *fh*, **unsigned char** *∗buf*, **unsigned** *len*, **int** *mode*)

Implementation

The *mode* argument is a bitmap describing the state of the file connected to *fh*, as for _sys_write().

Returns

The return value is one of the following:

• The number of characters *not* read (that is, *len - result* were read).

• An error indication.

• An EOF indicator. The EOF indication involves the setting of 0x80000000 in the normal result. The target-independent code is capable of handling either:

   Early EOF   The last read from a file returns some characters plus an EOF indicator.

   Late EOF   The last read returns only EOF.

5.11.6   _sys_write()

Writes the contents of a buffer to a file previously opened with _sys_open().

Syntax

`int _sys_write(FILEHANDLE` *fh*`, const unsigned char *`*buf*`, unsigned` *len*`, int` *mode*`)`

Implementation

The *mode* parameter is a bitmap describing the state of the file connected to *fh*, whether it is a binary file, and how it is buffered. The mode bits might be important if the file is connected to a terminal device because they specify whether or not the device is to be used raw (for example, whether the terminal input must be echoed). See the _IOxxx constants in `stdio.h` for definitions of user-accessible mode bits.

The default semihosting implementation of _sys_write() does not pass the *mode* parameter, because it is not required by the SYS_WRITE (0x05) semihosting SWI. If you are retargeting the C library, and you require the *mode* parameter, you must re-implement sys_io.o.

Returns

The return value is either:

*   a positive number representing the number of characters *not* written (so any nonzero return value denotes a failure of some sort)

*   a negative number indicating an error.

5.11.7   **_sys_ensure()**

This function flushes buffers associated with a file handle.

Syntax

int _sys_ensure(FILEHANDLE *fh*)

Implementation

A call to _sys_ensure() flushes any buffers associated with file handle *fh*, and ensures that the file is up to date on the backing store medium.

Returns

If an error occurs, the result is negative.

5.11.8   **_sys_flen()**

This function returns the current length of a file.

Syntax

long _sys_flen(FILEHANDLE *fh*)

Implementation

The function is required to convert fseek(..., SEEK_END) into (..., SEEK_SET) as required by _sys_seek().

If fseek() is used with an underlying system that does not directly support seeking relative to the end of a file, _sys_flen() must be defined. If the underlying system can seek relative to the end of a file, you can define fseek() so that _sys_flen() is not required.

Returns

This function returns the current length of the file *fh*, or a negative error indicator.

5.11.9 _sys_seek()

This function puts the file pointer at offset *pos* from the beginning of the file.

Syntax

int _sys_seek(FILEHANDLE *fh*, **long** *pos*)

Implementation

This function sets the current read or write position to the new location *pos* relative to the start of the current file *fh*.

Returns

The result is non-negative if no error occurs or is negative if an error occurs.

5.11.10 _sys_istty()

This function determines if a file handle identifies a terminal.

Syntax

int _sys_istty(FILEHANDLE fh)

Implementation

When a file is connected to a terminal device, this function is used to provide unbuffered behavior by default (in the absence of a call to set(v)buf) and to prohibit seeking.

Returns

The return value is:

0	There is no interactive device
1	There is an interactive device
other	An error occurred.

5.11.11 _sys_tmpnam()

This function converts the file number *fileno* for a temporary file to a unique filename, for example tmp0001.

Syntax

void _sys_tmpnam(**char** *name, **int** *fileno*, **unsigned** *maxlength*)

Implementation

The function must be defined if tmpnam() or tmpfile() is used.

Returns

Returns the filename in *name*.

5.11.12 _sys_command_string()

This function retrieves the command line used to invoke the current application from the environment that called the application.

Syntax

char *_sys_command_string(**char** *cmd, **int** *len*)

where:

cmd   Is a pointer to a buffer that can be used to store the command line. It is
            not required that the command line is stored in *cmd*.

len   Is the length of the buffer.

Implementation

This function is called by the library startup code to set up argv and argc to pass to main().

Returns

The function must return either:

- A pointer to the command line, if successful. This can be either a pointer to the *cmd* buffer if it is used, or a pointer to wherever else the command line is stored.

- NULL, if not successful.

5.12   Tailoring other C library functions

This section describes target-dependent ISO C library functions. Implementation of these ISO standard functions depends entirely on the target operating system.

Table 5-17 lists all the target-dependent functions. None of these functions is used internally by the library. So if any of these functions are not implemented, only those applications that call the function directly will fail.

Table 5-17 ISO C library functions

Function	Description
clock() and _clock_init()	*clock()* on page 5-107 and *_clock_init()* on page 5-107
time()	*time()* on page 5-108
remove()	*remove()* on page 5-108
rename()	*rename()* on page 5-109
system()	*system()* on page 5-109
getenv()	*getenv()* on page 5-110
__getenv_init()	*_getenv_init()* on page 5-110

The default implementation of these functions is semihosted. That is, each function uses the semihosting SWI.

clock() and _clock_init() must be re-implemented together or not at all.

5.12.1   clock()

This is the standard C library clock function from `time.h`.

Syntax

`clock_t clock(`**`void`**`)`

Implementation

If the units of `clock_t()` differ from the default of centiseconds you must define `__CLK_TCK` on the compiler command line or in your own header file. The value in the definition is used for `CLK_TCK` and `CLOCKS_PER_SEC` (the default value is 100 for centiseconds). If you re-implement `clock()` you must also re-implement `_clock_init()`.

Returns

The returned value is an unsigned integer.

5.12.2   _clock_init()

This is an optional initialization function for `clock()`.

Syntax

`__weak `**`void`**` _clock_init(`**`void`**`)`

Implementation

You must provide a clock initialization function if `clock()` must work with a read-only timer. If implemented, `_clock_init()` is called from the library initialization code.

5.12.3   time()

This is the standard C library time() function from time.h.

Syntax

time_t time(time_t *timer)

The return value is an approximation of the current calendar time.

Returns

The value (time_t*)-1 is returned if the calendar time is not available. If timer is not a NULL pointer, the return value is also assigned to the time_t*.

5.12.4   remove()

This is the standard C library remove() function from stdio.h.

Syntax

int remove(**const char** *filename)

Implementation

remove() causes the file whose name is the string pointed to by *filename* to be removed. Subsequent attempts to open the file result in failure, unless it is created again. If the file is open, the behavior of the remove function is implementation-defined.

Returns

Returns zero if the operation succeeds or nonzero if it fails.

5.12.5   rename()

This is the standard C library rename() function from stdio.h.

Syntax

int rename(**const char** *∗old*, **const char** *∗new*)

Implementation

rename() causes the file whose name is the string pointed to by *old* to be subsequently known by the name given by the string pointed to by *new*. The file named *old* is effectively removed. If a file named by the string pointed to by *new* exists prior to the call of the rename function, the behavior is implementation-defined.

Returns

Returns zero if the operation succeeds or nonzero if it fails. If nonzero and the file existed previously it is still known by its original name.

5.12.6   system()

This is the standard C library system() function from stdlib.h.

Syntax

int system(**const char** *∗string*)

Implementation

system() passes the string pointed to by *string* to the host environment to be executed by a command processor in an implementation-defined manner. A null pointer can be used for *string*, to inquire whether a command processor exists.

Returns

If the argument is a null pointer, the system function returns nonzero only if a command processor is available.

If the argument is not a null pointer, the system function returns an implementation-defined value.

5.12.7   getenv()

This is the standard C library getenv() function from stdlib.h.

Syntax

char *getenv(**const char** *name*)

Implementation

The default implementation returns NULL indicating that no environment information is available. You can re-implement getenv() yourself. It depends on no other function and no other function depends on it.

If you redefine the function, you can also call a function _getenv_init(). The C library initialization code also calls this when the library is initialized, that is, before main() is entered.

The function searches the environment list, provided by the host environment, for a string that matches the string pointed to by *name*. The set of environment names and the method for altering the environment list are implementation-defined.

Returns

The return value is a pointer to a string associated with the matched list member. The array pointed to must not be modified by the program, but might be overwritten by a subsequent call to getenv().

5.12.8   _getenv_init()

This enables a user version of getenv() to initialize itself.

Syntax

void _getenv_init(**void**)

Implementation

If this function is defined, the C library initialization code calls it when the library is initialized, that is before main() is entered.

5.13   Selecting real-time division

The division helper routine supplied with the ARM libraries provides good overall performance. However, the amount of time required to perform a division depends on the input values. A 4-bit quotient requires only 12 cycles, but a 32-bit quotient requires 96 cycles. Some applications require a faster worst-case cycle count at the expense of lower average performance. For this reason, the ARM library provides two divide routines.

The real-time routine:

*   always executes in fewer than 45 cycles
*   is faster than the standard division helper routine for larger quotients
*   is slower than the standard division helper routine for typical quotients
*   returns the same results
*   calls the same error reporting mechanism on a division by zero
*   does not require any change in the surrounding code.

Select the real-time divide routine, instead of the generally more efficient routine, by using either:

*   `IMPORT __use_realtime_division` from assembly language
*   `#pragma import(__use_realtime_division)` from C.

———— **Note** ————

The real-time division routine works on any CPU that has long multiply instructions, that is ARM7TDMI® and later. However, division is significantly faster on ARMv5E, or later architectures.

If you reference `__use_realtime_division` and compile and link for a core that does not support long multiply instructions, the linker displays an error similar to:

```
Error: L6218E: Undefined symbol
__realtime_division_only_works_with_long_multiply_available__
(referred from myobj.o)
```

5.14   ISO implementation definition

This section describes how the libraries fulfill the requirements of the ISO specification, and includes:

*   *ISO C library implementation definition*
*   *Standard C++ library implementation definition* on page 5-118.

5.14.1   ISO C library implementation definition

The ISO specification leaves some details to the implementors, but requires that implementation choices be documented. This section describes the ARM library implementation.

In the generic ARM C library:

*   The macro `NULL` expands to the integer constant 0.

*   If a program redefines a reserved external identifier, an error might occur when the program is linked with the standard libraries. If it is not linked with standard libraries, no error is diagnosed.

*   The `assert()` function prints the following message and then calls the `abort()` function:

    `*** assertion failed: expression, file _FILE_, line _LINE_`

    Be aware that the default behavior for `assert()` depends on the optimization level you are using (see *Exiting from the program* on page 5-41 for details).

The following functions test for character values in the range `EOF` (–1) to 255 (inclusive):

*   `isalnum()`
*   `isalpha()`
*   `iscntrl()`
*   `islower()`
*   `isprint()`
*   `isupper()`
*   `ispunct()`.

The ISO C library variants are listed in *Library naming conventions* on page 5-128.

Mathematical functions

Table 5-18 shows how the mathematical functions respond when supplied with out-of-range arguments.

Table 5-18 Mathematical functions

Function	Condition	Returned value	Error number
acos(x)	abs(x) > 1	QNaN	EDOM
asin(x)	abs(x) > 1	QNaN	EDOM
atan2(x,y)	x =0, y = 0	QNaN	EDOM
atan2(x,y)	x = Inf, y = Inf	QNaN	EDOM
cos(x)	x=Inf	QNaN	EDOM
cosh(x)	Overflow	+Inf	ERANGE
exp(x)	Overflow	+Inf	ERANGE
exp(x)	Underflow	+0	ERANGE
fmod(x,y)	x=Inf	QNaN	EDOM
fmod(x,y)	y = 0	QNaN	EDOM
log(x)	x < 0	QNaN	EDOM
log(x)	x = 0	-Inf	EDOM
log10(x)	x < 0	QNaN	EDOM
log10(x)	x = 0	-Inf	EDOM
pow(x,y)	Overflow	+Inf	ERANGE
pow(x,y)	Underflow	0	ERANGE
pow(x,y)	x=0 or x=Inf, y=0	+1	EDOM
pow(x,y)	x=+0, y<0	-Inf	EDOM
pow(x,y)	x=-0, y<0 and y integer	-Inf	EDOM
pow(x,y)	x= -0, y<0 and y non-integer	QNaN	EDOM
pow(x,y)	x<0, y non-integer	QNaN	EDOM
pow(x,y)	x=1, y=Inf	QNaN	EDOM

Table 5-18 Mathematical functions

Function	Condition	Returned value	Error number
sqrt(x)	x < 0	QNaN	EDOM
sin(x)	x=Inf	QNaN	EDOM
sinh(x)	Overflow	+Inf	ERANGE
tan(x)	x=Inf	QNaN	EDOM
atan(x)	SNaN	SNaN	None
ceil(x)	SNaN	SNaN	None
floor(x)	SNaN	SNaN	None
frexp(x)	SNaN	SNaN	None
ldexp(x)	SNaN	SNaN	None
modf(x)	SNaN	SNaN	None
tanh(x)	SNaN	SNaN	None

HUGE_VAL is an alias for Inf. Consult the errno variable for the error number. Other than the cases shown in Table 5-18 on page 5-113, all functions return QNaN when passed QNaN and throw an invalid operation exception when passed SNaN.

Signal functions

Table 5-19 shows the signals supported by the signal() function.

Table 5-19 Signal functions

Signal	Number	Description	Additional argument
SIGABRT	1	This signal is only used if abort() or assert() are called by your application	None
SIGFPE	2	Used to signal any arithmetic exception, for example, division by zero. Used by hard and soft floating point and by integer division.	A set of bits from {FE_EX_INEXACT, FE_EX_UNDERFLOW, FE_EX_OVERFLOW, FE_EX_DIVBYZERO, FE_EX_INVALID, DIVBYZERO}
SIGILL	3	Illegal instruction	None
SIGINT	4	Attention request from user	None
SIGSEGV	5	Bad memory access	None
SIGTERM	6	Termination request	None
SIGSTAK	7	Stack overflow was detected (but only for code compiled with software stack checking ON).	None
SIGRTRED	8	Redirection failed on a runtime library input/output stream	Name of file or device being re-opened to redirect a standard stream
SIGRTMEM	9	Out of heap space during initialization or after corruption	Size of failed request
SIGUSR1	10	User-defined	User-defined
SIGUSR2	11	User-defined	User-defined
SIGPVFN	12	A pure virtual function was called from C++	-

Table 5-19 Signal functions

Signal	Number	Description	Additional argument
SIGCPPL	13	Exception from C++	-
SIGOUTOFHEAP	14	Returned by the C++ function `::operator new` when out of heap space	Size of failed request
reserved	15-31	Reserved	Reserved
other	> 31	User-defined	User-defined

A signal number greater than **SIGUSR2** can be passed through `__raise()`, and caught by the default signal handler, but it cannot be caught by a handler registered using `signal()`.

`signal()` returns an error code if you try to register a handler for a signal number greater than **SIGUSR2**.

The default handling of all recognized signals is to print a diagnostic message and call `exit()`. This default behavior applies at program startup and until you change it.

——— **Caution** ———

The IEEE 754 standard for floating-point processing states that the default action to an exception is to proceed without a trap. A raised exception in floating-point calculations does not, by default, generate **SIGFPE**. You can modify fp error handling by tailoring the functions and definitions in `fenv.h`. See *Tailoring error signaling, error handling, and program exit* on page 5-71 and Chapter 6 *Floating-point Support*.

For all the signals in Table 5-19 on page 5-115, when a signal occurs, if the handler points to a function, the equivalent of `signal(sig, SIG_DFL)` is executed before the call to the handler.

If the **SIGILL** signal is received by a handler specified to by the `signal()` function, the default handling is reset.

Input/output characteristics

The generic ARM C library has the following input/output characteristics:

- The last line of a text stream does not require a terminating newline character.

- Space characters written out to a text stream immediately before a newline character do appear when read back in.

- No null characters are appended to a binary output stream.

- The file position indicator of an append mode stream is initially placed at the end of the file.

- A write to a text stream causes the associated file to be truncated beyond the point where the write occurred if this is the behavior of the device category of the file.

- The characteristics of file buffering agree with section 4.9.3 of the ISO C standard. If semihosting is used, the maximum number of open files is limited by the available target memory.

- A zero-length file, that is, no characters have been written by an output stream, exists.

- A file can be opened many times for reading, but only once for writing or updating. A file cannot simultaneously be open for reading on one stream, and open for writing or updating on another.

- Local time zones and Daylight Saving Time are not implemented. The values returned indicate that the information is not available. For example, the `gmtime()` function always returns NULL.

- The status returned by `exit()` is the same value that was passed to it. For definitions of `EXIT_SUCCESS` and `EXIT_FAILURE`, see the header file `stdlib.h`. The semihosting SWI, however, does not pass the status back to the execution environment.

- The error messages returned by the `strerror()` function are identical to those given by the `perror()` function.

- If the size of area requested is zero, `calloc()`, `malloc()`, and `realloc()` return `NULL`.

- `abort()` closes all open files and deletes all temporary files.

- `fprintf()` prints `%p` arguments in lowercase hexadecimal format as if a precision of 8 had been specified. If the variant form (`%#p`) is used, the number is preceded by the character `@`.

- `fscanf()` treats `%p` arguments exactly the same as `%x` arguments.

- fscanf() always treats the character "-" in a %...[...] argument as a literal character.

- ftell() and fgetpos() set errno to the value of EDOM on failure.

- perror() generates the messages shown in Table 5-20.

Table 5-20 perror() messages

Error	Message
0	No error (errno = 0)
EDOM	EDOM - function argument out of range
ERANGE	ERANGE - function result not representable
ESIGNUM	ESIGNUM - illegal signal number
Others	Unknown error

The following characteristics must be specified in an ISO-compliant implementation. These are unspecified in the ARM C library:

- the validity of a filename
- whether remove() can remove an open file
- the effect of calling the rename() function when the new name already exists
- the effect of calling getenv() (the default is to return NULL, no value available)
- the effect of calling system()
- the value returned by clock().

5.14.2   Standard C++ library implementation definition

This section describes the implementation of the C++ libraries. The ARM C++ library provides all of the library defined in the *ISO/IEC 14822 :1998 International Standard for C++*, aside from some limitations described in Table 5-22 on page 5-120.

For information on implementation-defined behavior that is defined in the Rogue Wave C++ library, see the included Rogue Wave HTML documentation. By default, this is installed in *install_directory*\Documentation\RogueWave.

The standard C++ library is distributed in binary form only.

Table 5-21 shows the requirements that the C++ library places on the C library.

Table 5-21 C++ requirements on the C library

File	Required function in C library
ctype.h	isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), toupper()
locale.h	localeconv(), setlocale()
math.h	acos(), asin(), atan2(), atan(), ceil(), cos(), cosh(), exp(), fabs(), floor(), fmod(), frexp(), ldexp(), log10(), log(), modf(), pow(), sin(), sinh(), sqrt(), tan(), tanh()
setjmp.h	longjmp()
signal.h	raise(), signal()
stdio.h	clearerr(), fclose(), feof(), ferror(), fflush(), fgetc(), fgetpos(), fgets(), fopen(), fprintf(), fputc(), fputs(), fread(), freopen(), fscanf(), fseek(), fsetpos(), ftell(), fwrite(), getc(), getchar(), gets(), perror(), printf(), putc(), putchar(), puts(), remove(), rename(), rewind(), scanf(), setbuf(), setvbuf(), sprintf(), sscanf(), tmpfile(), tmpnam(), ungetc(), vfprintf(), vprintf(), vsprintf()
stdlib.h	abort(), abs(), atexit(), atof(), atoi(), atol(), bsearch(), calloc(), div(), exit(), free(), getenv(), labs(), ldiv(), malloc(), mblen(), qsort(), rand(), realloc(), srand(), strtod(), strtol(), strtoul(), system()
string.h	memchr(), memcmp(), memcpy(), memmove(), memset(), memset(), strcat(), strchr(), strcmp(), strcoll(), strcpy(), strcspn(), strerror(), strlen(), strncat(), strncmp(), strncpy(), strpbrk(), strrchr(), strspn(), strstr(), strtok(), strxfrm()
time.h	asctime(), clock(), ctime(), difftime(), mktime(), strftime(), time()

Table 5-22 describes the most important features missing from the current release.

Table 5-22 Standard C++ library differences

Standard	Implementation differences
locale	The locale message facet is not supported. It fails to open catalogs at runtime because the ARM C library does not support `catopen` and `catclose` through `nl_types.h`. One of two locale definitions can be selected at link time. Other locales can be created by user-redefinable functions.
Timezone	Not supported. The ARM C library does not support it.

5.15   C library extensions

This section describes C library extensions and functions. Some are defined by the C99 draft standard (*ISO/IEC 9899:1999E*) and some are specific to the ARM compiler. These are summarized in Table 5-23.

Table 5-23 C library extensions

Function	Header file definition	Extension
atoll() on page 5-122	stdlib.h	C99 standard
strtoll() on page 5-122	stdlib.h	C99 standard
strtoull() on page 5-122	stdlib.h	C99 standard
printf() on page 5-122	stdlib.h	C99 standard
snprintf() on page 5-123	stdio.h	C99 standard
vsnprintf() on page 5-123	stdio.h	C99 standard
lldiv() on page 5-123	stdlib.h	C99 standard
llabs() on page 5-124	stdlib.h	C99 standard
alloca() on page 5-124	alloca.h	Common nonstandard extension to many C libraries
_fisatty() on page 5-124	stdio.h	Specific to ARM compiler
__heapstats() on page 5-125	stdlib.h	Specific to ARM compiler
__heapvalid() on page 5-126	stdlib.h	Specific to ARM compiler

The headers <stdint.h> and <inttypes.h> from C99 are also available.

5.15.1   atoll()

The `atoll()` function converts a decimal string into an integer, similarly to the ISO functions `atol()` and `atoi()`, but returning a **long long** result. Like `atoi()`, `atoll()` can accept octal or hexadecimal input if the string begins with `0` or `0x`.

Syntax

long long atoll(**const char** *nptr*)

5.15.2   strtoll()

The `strtoll()` function converts a string in an arbitrary base to an integer. This is similar to the ISO function `strtol()`, but returns a **long long** result. Like `strtol()`, the parameter *endptr* can hold the location where a pointer to the end of the translated string is to be stored, or can be `NULL`. The parameter *base* must contain the number base. Setting *base* to zero indicates that the base is to be selected in the same way as `atoll()`.

Syntax

long long strtoll(**const char** *nptr*, **char** **endptr*, **int** *base*)

5.15.3   strtoull()

`strtoull()` is exactly the same as `strtoll()`, but returns an **unsigned long long**.

Syntax

unsigned long long strtoull(**const char** *nptr*, **char** **endptr*, **int** *base*)

5.15.4   printf()

`printf()` is fully conformant to the ISO C89 standard at all times. It also optionally supports the additional format directives defined in C99, that is, `%a` and `%A` for hexadecimal floating-point, and `%E`, `%F`, and `%G` as uppercase versions of `%e`, `%f`, and `%g`, but these are not included by default.

To enable the C99 features in `printf()` you must specify the pragma `#pragma import(__use_c99_library)` This affects all functions in the `printf()` and `scanf()` families.

Syntax

int printf(**const char** *format*, ...)

5.15.5   snprintf()

The snprintf() function works almost exactly like the ISO sprintf() function, except that the caller can specify the maximum size of the buffer. The return value is the length of the complete formatted string that would have been written if the buffer were big enough. Therefore, the string written into the buffer is complete only if the return value is at least zero and at most n-1.

The *bufsize* parameter specifies the number of characters of *buffer* that the function can write into, *including* the terminating null.

<stdio.h> is an ISO header file, but the function is prohibited by the ISO C library standard. It is therefore not available if you use the compiler with the --strict option.

Syntax

int snprintf(**char** **buffer*, **size_t** *bufsize*, **const char** **format*, ...)

5.15.6   vsnprintf()

The vsnprintf() function works almost exactly like the ISO vsprintf() function, except that the caller can specify the maximum size of the buffer. The return value is the length of the complete formatted string that would have been written if the buffer were big enough. Therefore, the string written into the buffer is complete only if the return value is at least zero and at most n-1.

The *bufsize* parameter specifies the number of characters of *buffer* that the function can write into, *including* the terminating null.

<stdio.h> is an ISO header file, but the function is prohibited by the ISO C library standard. It is therefore not available if you use the compiler with the --strict option.

Syntax

int vsnprintf(**char** **buffer*, size_t *bufsize*, **const char** **format*, va_list *ap*)

5.15.7   lldiv()

The lldiv function divides two **long long** integers and returns both the quotient and the remainder. It is the **long long** equivalent of the ISO function ldiv. The return type lldiv_t is a structure containing two **long long** members, called quot and rem.

<stdlib.h> is an ISO header file, but the function is prohibited by the ISO C library standard. It is therefore not available if you use the compiler with the --strict option.

Syntax

lldiv_t lldiv(**long long** *num*, **long long** *denom*)

5.15.8   llabs()

The llabs() returns the absolute value of its input. It is the **long long** equivalent of the ISO function labs.

<stdlib.h> is an ISO header file, but the function is prohibited by the ISO C library standard. It is therefore not available if you use the compiler with the --strict option.

Syntax

long long llabs(**long long** *num*)

5.15.9   alloca()

The alloca() function allocates local storage in a function. It returns a pointer to *size* bytes of memory, or NULL if not enough memory was available. The default implementation returns an eight-byte aligned block of memory.

Memory returned from alloca() must never be passed to free(). Instead, the memory is deallocated automatically when the function that called alloca() returns.

alloca() must not be called through a function pointer. You must take care when using alloca() and setjmp() in the same function, because memory allocated by alloca() between calling setjmp() and longjmp() is deallocated by the call to longjmp().

This function is a common nonstandard extension to many C libraries.

Syntax

void∗ alloca(size_t *size*)

5.15.10   _fisatty()

The _fisatty() function determines whether the given stdio stream is attached to a terminal device or a normal file. It calls the _sys_istty() low-level function (see *Tailoring the input/output functions* on page 5-95) on the underlying file handle.

This function is an extension that is specific to the ARM library.

Syntax

int _fisatty(FILE ∗*stream*)

The return value indicates the stream destination:

0   A file.

1   A terminal.

Negative   An error.

5.15.11 __heapstats()

The __heapstats() function displays statistics on the state of the storage allocation heap. It calls the __Heap_Stats() function, that you can re-implement if you choose to do your own storage management (see *__Heap_Stats()* on page 5-84). The default implementation in the ARM compiler gives information on how many free blocks exist, and estimates their size ranges.

Example 5-20 shows an example of the output from __heapstats(). Line 1 of the output displays the total number of bytes, the number of free blocks, and the average size. The following lines give an estimate the size of each block in bytes, expressed as a range. __heapstats() does not give information on the number of used blocks.

Example 5-20 Output from __heapstats()

```
32272 bytes in 2 free blocks (avge size 16136)
1 blocks 2^12+1 to 2^13
1 blocks 2^13+1 to 2^14
```

The function outputs its results by calling the output function *dprint*, that must work like fprintf(). The first parameter passed to *dprint* is the supplied pointer *param*. You can pass fprintf() itself, provided you cast it to the right function pointer type. This type is defined as a **typedef** for convenience. It is called __heapprt. For example:

```
__heapstats((__heapprt)fprintf, stderr);
```

——— **Note** ———

If you call fprintf() on a stream that you have not already sent output to, the library calls malloc() internally to create a buffer for the stream. If this happens in the middle of a call to __heapstats(), the heap might be corrupted. Therefore, you must ensure you have already sent some output to stderr in the this example.

If you are using the default single-region memory model, heap memory is allocated only as it is required. This means that the amount of free heap changes as you allocate and deallocate memory. For example, the sequence:

```
int *ip;
__heapstats((__heapprt)fprintf,stderr);    // print initial free heap size
ip = malloc(200000);
free(ip);
__heapstats((__heapprt)fprintf,stderr);     // print heap size after freeing
```

gives output such as:

```
4076 bytes in 1 free blocks (avge size 4076)
1 blocks 2^10+1 to 2^11
2008180 bytes in 1 free blocks (avge size 2008180)
1 blocks 2^19+1 to 2^20
```

This function is an extension that is specific to the ARM library.

Syntax

void __heapstats(**int** (*dprint*)(**void** *param*, **char const** *format*,...), **void***
              param)

5.15.12 __heapvalid()

The __heapvalid() function performs a consistency check on the heap. It outputs detailed information about every free block if the *verbose* parameter is nonzero, and only output errors otherwise.

The function outputs its results by calling the output function *dprint*, that must work like fprintf(). The first parameter passed to *dprint* is the supplied pointer *param*. You can pass fprintf() itself, provided you cast it to the right function pointer type. This type is defined as a **typedef** for convenience. It is called __heapprt. For example:

Example 5-21 Calling __heapvalid() with fprintf()

```
__heapvalid((__heapprt) fprintf, stderr, 0);
```

If you call fprintf() on a stream that you have not already sent output to, the library calls malloc() internally to create a buffer for the stream. If this happens in the middle of a call to __heapvalid(), the heap might be corrupted. You must therefore ensure you have already sent some output to stderr. The code in Example 5-21 fails if you have not already written to the stream.

This function is an extension that is specific to the ARM library.

Syntax

```
int __heapvalid(int (*dprint)( void *param, char const *format,...), void*
                param, int verbose)
```

5.16   Library naming conventions

The library filename identifies how the variant was built as follows:

root_<arch><fpu><stack><entrant>.<endian>

The values for the fields of the filename, and the relevant build options are:

root	c	ISO C and C++ basic runtime support.
	f	IEEE compliant library with a fixed rounding mode (Round to nearest) and no inexact exceptions.
	fj	IEEE compliant library with a fixed rounding mode (Round to nearest) and no exceptions (`--fpmode ieee_no_fenv`).
	fz	Behaves like the fj library, but additionally flushes denormals and infinities to zero (`--fpmode std` and `--fpmode fast`). This library behaves like the ARM VFP in Fast mode. This is the default.
	g	IEEE compliant library with configurable rounding mode and all IEEE exceptions.
	m	Transcendental math functions.
	h	Compiler support (helper) library.
	cpp	Rogue Wave C++ library.
	cpprt	The ARM C++ runtime libraries.
arch	a	An ARM library.
	t	A Thumb library (`--apcs /interwork`). See *Thumb C libraries* on page 5-5.
fpu	_p	Soft VFP (`--fpu softvfp`).
	__	Does not use floating-point instructions (`--fpu none`).
stack	u	Does not use software stack checking (`--apcs /noswst`).
	s	Uses software stack checking (`--apcs /swst`).
	_	Not applicable.
entrant	n	Static data accessed position dependently (`--apcs /norwpi`).
	e	Static data accessed position independently (`--apcs /rwpi`).
	f	FPIC addressing is enabled (`--apcs /fpic`).
	_	Not applicable.
endian	l	Little-endian (`--li`).
	b	Big-endian (`--bi`).

For example:

c_a__un.l   ARM library, no stack checking, not reentrant, and little-endian.

f_a   ARM library, used with --fpu none.

fj_tvp   Thumb library, VFP.

m_tvpu   Thumb library, VFP, and no stack checking

cpp_t_pse   High-level C++ functions that do not require FP arithmetic, Thumb, Soft
             VFP, pure-endian, RWPI, stack checking

cpprt_a__s   ARM C++ runtime library, with --fpu none and stack checking

Also, see *Specifying the target processor or architecture* on page 2-39 for details on
how to specify a specific architecture or processor.

——— **Note** ———

Not all variant name combinations are valid. See the armlib and cpplib directories for
the libraries that are supplied with RVCT.

Chapter 6
Floating-point Support

This chapter describes the ARM® support for floating-point computations. It contains the following sections:

- *About floating-point support* on page 6-2
- *The software floating-point library, fplib* on page 6-3
- *Controlling the floating-point environment* on page 6-10
- *The math library, mathlib* on page 6-27
- *IEEE 754 arithmetic* on page 6-36.

6.1 About floating-point support

The ARM floating-point environment is an implementation of the IEEE 754 standard for binary floating-point arithmetic. See *IEEE 754 arithmetic* on page 6-36 for details of the ARM implementation of the standard.

An ARM system might have:

* a *Vector Floating-Point* (VFP) coprocessor
* no floating-point hardware.

———— **Note** ————

RVDK v2.2 does not support any floating-point hardware.

The compilation tools only accept `-fpu none` or `-fpu softvfp` (default).

6.2 The software floating-point library, fplib

When programs are compiled to use a floating-point coprocessor, they perform basic floating-point arithmetic (for example addition and multiplication) by means of floating-point machine instructions for the target coprocessor. When programs are compiled to use software floating-point, there is no floating-point instruction set available, and so the ARM libraries must provide a set of procedure calls to do floating-point arithmetic. These procedures are in the software floating-point library, fplib.

This section describes:

• *Features of the floating-point library, fplib*
• *Arithmetic on numbers in a particular format* on page 6-4
• *Conversions between floats, doubles, and ints* on page 6-6
• *Conversions between long longs and other number formats* on page 6-7
• *Floating-point comparisons* on page 6-8
• *C99 functions* on page 6-9.

6.2.1 Features of the floating-point library, fplib

Floating-point routines have names like _dadd (add two **double**s) and _fdiv (divide two **float**s). User programs can call these routines directly. Even in environments with a coprocessor, the routines are provided. However, they are typically only a few instructions long (because all they do is to execute the appropriate coprocessor instruction).

All the fplib routines are called using a software floating-point variant of the calling standard. This means that floating-point arguments are passed and returned in integer registers. In the rest of the program, if the program is compiled for a coprocessor, floating-point data is passed in its floating-point registers.

So, for example, _dadd takes a **double** in registers r0 and r1, and another **double** in registers r2 and r3, and returns the sum in r0 and r1.

———— **Note** ————

For a **double** in registers r0 and r1, the register that holds the high 32 bits of the **double** depends on whether your program is little-endian or big-endian.

————————————

C programs are not required to handle the register allocation.

All the fplib routines (except those listed in Table 6-5 on page 6-9) are declared in the header file `rt_fp.h`. You can include this file if you want to call an fplib routine directly. The routines shown in Table 6-5 on page 6-9 are declared in the standard header file `math.h`.

To call a function from assembler, the softfp function is called `__softfp_`*fn*. For example, to call the `cos()` function, do the following:

```
IMPORT __softfp_cos
BL __softfp_cos
```

For a complete list of fplib routines see:
- Table 6-1
- Table 6-2 on page 6-6
- Table 6-3 on page 6-7
- Table 6-4 on page 6-8
- Table 6-5 on page 6-9.

6.2.2 Arithmetic on numbers in a particular format

Table 6-1 describes routines to perform arithmetic on numbers in a particular format. Arguments and results are always in the same format.

Table 6-1 Arithmetic routines

Function	Argument types	Result type	Operation
_fadd	2 **float**	**float**	Return x plus y
_fsub	2 **float**	**float**	Return x minus y
_frsb	2 **float**	**float**	Return y minus x
_fmul	2 **float**	**float**	Return x times y
_fdiv	2 **float**	**float**	Return x divided by y
_frdiv	2 **float**	**float**	Return y divided by x
_frem	2 **float**	**float**	Return remainder[a] of x by y
_frnd	**float**	**float**	Return x rounded to an integer[b]
_fsqrt	**float**	**float**	Return square root of x
_dadd	2 **double**	**double**	Return x plus y

Table 6-1 Arithmetic routines

Function	Argument types	Result type	Operation
_dsub	2 **double**	**double**	Return x minus y
_drsb	2 **double**	**double**	Return y minus x
_dmul	2 **double**	**double**	Return x times y
_ddiv	2 **double**	**double**	Return x divided by y
_drdiv	2 **double**	**double**	Return y divided by x
_drem	2 **double**	**double**	Return remainder[a] of x by y
_drnd	**double**	**double**	Return x rounded to an integer[b]
_dsqrt	**double**	**double**	Return square root of x

a. Functions that perform the IEEE 754 remainder operation. This is defined to take two numbers, x and y, and return a number z so that $z = x - n * y$, where n is an integer. To return an exactly correct result, n is chosen so that z is no bigger than half of x (so that z might be negative even if both x and y are positive). The IEEE 754 remainder function is not the same as the operation performed by the C library function fmod, where z always has the same sign as x. Where the IEEE 754 specification gives two acceptable choices of n, the even one is chosen. This behavior occurs independently of the current rounding mode.

b. Functions that perform the IEEE 754 round-to-integer operation. This takes a number and rounds it to an integer (in accordance with the current rounding mode), but returns that integer in the floating-point number format rather than as a C **int** variable. To convert a number to an **int** variable, you must use the _ffix routines described in Table 6-2 on page 6-6

6.2.3 Conversions between floats, doubles, and ints

Table 6-2 describes routines to perform conversions between number formats, excluding `long long`s.

Table 6-2 Number format conversion routines

Function	Argument types	Result type
_f2d	**float**	**double**
_d2f	**double**	**float**
_fflt	**int**	**float**
_ffltu	**unsigned int**	**float**
_dflt	**int**	**double**
_dfltu	**unsigned int**	**double**
_ffix	**float**	**int**[a]
_ffix_r	**float**	**int**
_ffixu	**float**	**unsigned int**[a]
_ffixu_r	**float**	**unsigned int**
_dfix	**double**	**int**[a]
_dfix_r	**double**	**int**
_dfixu	**double**	**unsigned int**[a]
_dfixu_r	**double**	**unsigned int**

a. Rounded toward zero, independently of the current rounding mode. This is because the C standard requires implicit conversions to integers to round this way, so it is convenient not to have to change the rounding mode to do so. Each function has a corresponding function with _r on the end of its name, that performs the same operation but rounds according to the current mode.

6.2.4 Conversions between long longs and other number formats

Table 6-3 describes routines to perform conversions between **long long**s and other number formats.

Table 6-3 Conversion routines involving long long format

Function	Argument types	Result type
_ll_sto_f	**long long**	**float**
_ll_uto_f	**unsigned long long**	**float**
_ll_sto_d	**long long**	**double**
_ll_uto_d	**unsigned long long**	**double**
_ll_sfrom_f	**float**	**long long**[a]
_ll_sfrom_f_r	**float**	**long long**
_ll_ufrom_f	**float**	**unsigned long long**[a]
_ll_ufrom_f_r	**float**	**unsigned long long**
_ll_sfrom_d	**double**	**long long**[a]
_ll_sfrom_d_r	**double**	**long long**
_ll_ufrom_d	**double**	**unsigned long long**[a]
_ll_ufrom_d_r	**double**	**unsigned long long**

a. Rounded toward zero, independently of the current rounding mode. This is because the C standard requires implicit conversions to integers to round this way, so it is convenient not to have to change the rounding mode to do so. Each function has a corresponding function with _r on the end of its name, that performs the same operation but rounds according to the current mode.

6.2.5 Floating-point comparisons

Table 6-4 describes routines to perform comparisons between floating-point numbers.

Table 6-4 Floating-point comparison routines

Function	Argument types	Result type	Condition tested
_fcmpeq	2 **float**	Flags, EQ/NE	x equal to y[a]
_fcmpge	2 **float**	Flags, HS/LO	x greater than or equal to y[a,b]
_fcmple	2 **float**	Flags, HI/LS	x less than or equal to y[a,b]
_feq	2 **float**	Boolean	x equal to y
_fneq	2 **float**	Boolean	x not equal to y
_fgeq	2 **float**	Boolean	x greater than or equal to y[b]
_fgr	2 **float**	Boolean	x greater than y[b]
_fleq	2 **float**	Boolean	x less than or equal to y[b]
_fls	2 **float**	Boolean	x less than y[b]
_dcmpeq	2 **double**	Flags, EQ/NE	x equal to y[a]
_dcmpge	2 **double**	Flags, HS/LO	x greater than or equal to y[a,b]
_dcmple	2 **double**	Flags, HI/LS	x less than or equal to y[a,b]
_deq	2 **double**	Boolean	x equal to y
_dneq	2 **double**	Boolean	x not equal to y
_dgeq	2 **double**	Boolean	x greater than or equal to y[b]
_dgr	2 **double**	Boolean	x greater than y[b]
_dleq	2 **double**	Boolean	x less than or equal to y[b]
_dls	2 **double**	Boolean	x less than y[b]

a. Returns results in the ARM condition flags. This is efficient in assembly language, because you can directly follow a call to the function with a conditional instruction, but it means there is no way to use these functions from C. These functions are not declared in rt_fp.h.

b. Causes an Invalid Operation exception if either argument is a NaN, even a quiet NaN. Other functions only cause Invalid Operation if an argument is an SNaN. QNaNs return *not equal* when compared to anything, including other QNaNs (so comparing a QNaN to the same QNaN still returns *not equal*).

6.2.6   C99 functions

Table 6-5 describes routines that implement C99 functionality.

Table 6-5 C99 routines

Function	Argument types	Result type	Returns section	Standard
ilogb	**double**	**int**	Exponent of argument x	7.12.6.5
ilogbf	**float**	**int**	Exponent of argument x	7.12.6.5
ilogbl	**long double**	**int**	Exponent of argument x	7.12.6.5
logb	**double**	**double**	Exponent of argument x	7.12.6.11
logbf	**float**	**float**	Exponent of argument x	7.12.6.11
logbl	**long double**	**long double**	Exponent of argument x	7.12.6.11
scalbn	**double**, **int**	**double**	x * (FLT_RADIX ** n)	7.12.6.13
scalbnf	**float**, **int**	**float**	x * (FLT_RADIX ** n)	7.12.6.13
scalbnl	**long double**, **int**	**long double**	x * (FLT_RADIX ** n)	7.12.6.13
scalbln	**double**, **long int**	**double**	x * (FLT_RADIX ** n)	7.12.6.13
scalblnf	**float**, **long int**	**float**	x * (FLT_RADIX ** n)	7.12.6.13
scalblnl	**long double**, **long int**	**long double**	x * (FLT_RADIX ** n)	7.12.6.13
nextafter	2 **double**	**double**	Next representable value after x towards y	7.12.11.3
nextafterf	2 **float**	**float**	Next representable value after x towards y	7.12.11.3
nextafterl	2 **long double**	**long double**	Next representable value after x towards y	7.12.11.3
nexttoward	**double**, **long double**	**double**	Next representable value after x towards y	7.12.11.4
nexttowardf	**float**, **long double**	**float**	Next representable value after x towards y	7.12.11.4
nexttowardl	2 **long double**	**long double**	Next representable value after x towards y	7.12.11.4

6.3 Controlling the floating-point environment

This section describes the functions you can use to control the ARM floating-point environment. The ARM compiler supplies several different interfaces to the floating-point environment, for compatibility and porting ease. With these functions, you can change the rounding mode, enable and disable trapping of exceptions, and install your own custom exception trap handlers.

This section includes:

- *__ieee_status()*
- *__fp_status()* on page 6-13
- *Microsoft compatibility functions* on page 6-15
- *C9X-compatible functions* on page 6-17
- *ARM compiler extensions to the C9X interface* on page 6-20.

6.3.1 __ieee_status()

The ARM compiler supports an interface to the status word in the floating-point environment. This is called `__ieee_status` and it is generally the most efficient function to use for modifying the status word for VFP. `__ieee_status` is defined in `fenv.h`.

`__ieee_status` has the prototype:

```
unsigned int __ieee_status(unsigned int mask,
                           unsigned int flags);
```

The function modifies the writable parts of the status word according to the parameters, and returns the previous value of the whole word.

The writable bits are modified by setting them to

```
new = (old & ~mask) ^ flags;
```

Four different operations can be performed on each bit of the status word, depending on the corresponding bits in mask and flags (see Table 6-6).

Table 6-6 Status word bit modification

Bit of mask	Bit of flags	Effect
0	0	Leave alone

Table 6-6 Status word bit modification

Bit of mask	Bit of flags	Effect
0	1	Toggle
1	0	Set to 0
1	1	Set to 1

The layout of the status word as seen by `__ieee_status` is shown in Figure 6-1.

31	25 24	23 22	21 20	19 18	16 15	13 12	8 7	5 4	0
R	FZ	RM	VFP	R	VFP	R	Masks	R	Sticky
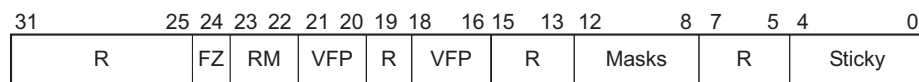
Figure 6-1 IEEE status word layout

The fields in Figure 6-1 are as follows:

- Bits 0 to 4 (values `0x1` to `0x10`, respectively) are the sticky flags, or cumulative flags, for each exception. The sticky flag for an exception is set to 1 whenever that exception happens and is not trapped. Sticky flags are never cleared by the system, only by the user. The mapping of exceptions to bits is:

  — bit 0 (`0x01`) is for the Invalid Operation exception

  — bit 1 (`0x02`) is for the Divide by Zero exception

  — bit 2 (`0x04`) is for the Overflow exception

  — bit 3 (`0x08`) is for the Underflow exception

  — bit 4 (`0x10`) is for the Inexact Result exception.

- Bits 8 to 12 (values `0x100` to `0x1000`) are the exception masks. These control whether each exception is trapped or not. If a bit is set to 1, the corresponding exception is trapped. If a bit is set to 0, the corresponding exception sets its sticky flag and return a plausible result, as described in *Exceptions* on page 6-40.

- Bits 16 to 18, and bits 20 and 21, are used by VFP to control the VFP vector capability. The `__ieee_status` call does not let you modify these bits.

• Bits 22 and 23 control the rounding mode (Table 6-7).

Table 6-7 Rounding mode control

Bits	Rounding mode
00	Round to nearest
01	Round up
10	Round down
11	Round toward zero

―――― **Note** ――――

The standard fplib libraries f∗ support only the Round to nearest rounding mode. If you require support for the other rounding modes, you must use the full IEEE g∗ libraries. See *Library naming conventions* on page 5-128.

• Bit 24 enables FZ (Flush to Zero) mode if it is set. In FZ mode, denormals are forced to zero to speed up processing (because denormals can be difficult to work with and slow down floating-point systems). Setting this bit reduces accuracy but might increase speed.

―――― **Note** ――――

The standard fplib libraries do not support the FZ mode. Instead, each library either always flushes to zero or never flushes to zero, and you choose the library to use at build time.

• Bits marked R are reserved. They cannot be written to by the __ieee_status call, and you must ignore anything you find in them

In addition to defining the __ieee_status call itself, fenv.h also defines some constants to be used for the arguments:

```
#define FE_IEEE_FLUSHZERO          (0x01000000)
#define FE_IEEE_ROUND_TONEAREST    (0x00000000)
#define FE_IEEE_ROUND_UPWARD       (0x00400000)
#define FE_IEEE_ROUND_DOWNWARD     (0x00800000)
#define FE_IEEE_ROUND_TOWARDZERO   (0x00C00000)
#define FE_IEEE_ROUND_MASK         (0x00C00000)
#define FE_IEEE_MASK_INVALID       (0x00000100)
#define FE_IEEE_MASK_DIVBYZERO     (0x00000200)
#define FE_IEEE_MASK_OVERFLOW      (0x00000400)
```

```
#define FE_IEEE_MASK_UNDERFLOW    (0x00000800)
#define FE_IEEE_MASK_INEXACT      (0x00001000)
#define FE_IEEE_MASK_ALL_EXCEPT   (0x00001F00)
#define FE_IEEE_INVALID           (0x00000001)
#define FE_IEEE_DIVBYZERO         (0x00000002)
#define FE_IEEE_OVERFLOW          (0x00000004)
#define FE_IEEE_UNDERFLOW         (0x00000008)
#define FE_IEEE_INEXACT           (0x00000010)
#define FE_IEEE_ALL_EXCEPT        (0x0000001F)
```

For example, to set the rounding mode to round down, you would do:

```
__ieee_status(FE_IEEE_ROUND_MASK, FE_IEEE_ROUND_DOWNWARD);
```

To trap the Invalid Operation exception and untrap all other exceptions:

```
__ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_INVALID);
```

To untrap the Inexact Result exception:

```
__ieee_status(FE_IEEE_MASK_INEXACT, 0);
```

To clear the Underflow sticky flag:

```
__ieee_status(FE_IEEE_UNDERFLOW, 0);
```

6.3.2   __fp_status()

Previous versions of the ARM libraries implemented a function called __fp_status that manipulated a status word in the floating-point environment. This is the same as __ieee_status but uses an older-style status word layout. The ARM compiler still supports the __fp_status function for backwards compatibility. __fp_status is defined in stdlib.h.

__fp_status has the following prototype:

unsigned int __fp_status(**unsigned int** mask, **unsigned int** flags);

The layout of the status word as seen by __fp_status is shown in Figure 6-2.

31	24 23	21 20	16 15	13 12	8 7	5 4	0
System ID	R	Masks	R	FPA only	R	Sticky	
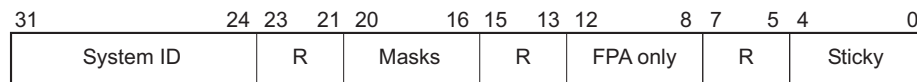
Figure 6-2 Floating-point status word layout

The fields in Figure 6-2 on page 6-13 are as follows:

- Bits 0 to 4 are the sticky flags, as described in *__ieee_status()* on page 6-10.

- Bits 8 to 12 (values `0x100` to `0x1000`) control various aspects of the FPA floating-point. Any attempt to write to these bits is ignored if there is no FPA on your system.

- Bits 16 to 20 (values `0x10000` to `0x100000`) are the exception masks, as described in *__ieee_status()* on page 6-10, but in a different place.

- Bits 24 to 31 contain the system ID that cannot be changed. It is set to `0x40` for software floating-point.

- Bits marked `R` are reserved. They cannot be written to by the `__fp_status` call, and you must ignore anything you find in them.

The rounding mode cannot be changed with the `__fp_status` call.

In addition to defining the `__fp_status` call itself, `stdlib.h` also defines some constants to be used for the arguments:

```
#define __fpsr_IXE  0x100000
#define __fpsr_UFE  0x80000
#define __fpsr_OFE  0x40000
#define __fpsr_DZE  0x20000
#define __fpsr_IOE  0x10000
#define __fpsr_IXC  0x10
#define __fpsr_UFC  0x8
#define __fpsr_OFC  0x4
#define __fpsr_DZC  0x2
#define __fpsr_IOC  0x1
```

For example, to trap the Invalid Operation exception and untrap all other exceptions, you would do:

```
__fp_status(_fpsr_IXE | _fpsr_UFE | _fpsr_OFE |
            _fpsr_DZE | _fpsr_IOE, _fpsr_IOE);
```

To untrap the Inexact Result exception:

```
__fp_status(_fpsr_IXE, 0);
```

To clear the Underflow sticky flag:

```
__fp_status(_fpsr_UFC, 0);
```

6.3.3 Microsoft compatibility functions

The following functions are implemented for compatibility with Microsoft products, to ease porting of floating-point code to the ARM architecture. They are defined in `float.h`.

_controlfp()

The function `_controlfp` enables you to control exception traps and rounding modes:

```
unsigned int _controlfp(unsigned int new, unsigned int mask);
```

This function also modifies a control word using a mask to isolate the bits to modify. For every bit of `mask` that is zero, the corresponding control word bit is unchanged. For every bit of `mask` that is nonzero, the corresponding control word bit is set to the value of the corresponding bit of `new`. The return value is the previous state of the control word.

———— **Note** ————

This is not quite the same as the behavior of `__ieee_status` (or `__fp_status`), where you can toggle a bit by setting a zero in the mask word and a one in the flags word.

Table 6-8 describes the macros you can use to form the arguments to `_controlfp`.

Table 6-8 _controlfp argument macros

Macro	Description
_MCW_EM	Mask containing all exception bits
_EM_INVALID	Bit describing the Invalid Operation exception
_EM_ZERODIVIDE	Bit describing the Divide by Zero exception
_EM_OVERFLOW	Bit describing the Overflow exception
_EM_UNDERFLOW	Bit describing the Underflow exception
_EM_INEXACT	Bit describing the Inexact Result exception
_MCW_RC	Mask for the rounding mode field
_RC_CHOP	Rounding mode value describing Round Toward Zero

Table 6-8 _controlfp argument macros

Macro	Description
_RC_UP	Rounding mode value describing Round Up
_RC_DOWN	Rounding mode value describing Round Down
_RC_NEAR	Rounding mode value describing Round To Nearest

———— **Note** ————

It is not guaranteed that the values of these macros will remain the same in future versions of ARM products. To ensure that your code continues to work if the value changes in future releases, use the macro rather than its value.

————————

For example, to set the rounding mode to round down, you would do:

```
_controlfp(_RC_DOWN, _MCW_RC);
```

To trap the Invalid Operation exception and untrap all other exceptions:

```
_controlfp(_EM_INVALID, _MCW_EM);
```

To untrap the Inexact Result exception:

```
_controlfp(0, _EM_INEXACT);
```

_clearfp()

The function _clearfp clears all five exception sticky flags, and returns their previous value. The macros given in Table 6-8 on page 6-15, for example _EM_INVALID, _EM_ZERODIVIDE, can be used to test bits of the returned result.

_clearfp has the following prototype:

unsigned _clearfp(**void**);

The _statusfp function

The function _statusfp returns the current value of the exception sticky flags. You can use the macros shown in Table 6-8 on page 6-15 to test bits of the returned result, for example, _EM_INVALID, _EM_ZERODIVIDE.

_statusfp has the following prototype:

```
unsigned _statusfp(void);
```

6.3.4   C9X-compatible functions

The ARM compiler also supports a set of functions defined in the C9X draft standard, in addition to the functions described previously.

These C9X-compatible functions are the only interface that enables you to install custom exception trap handlers with the ability to invent a return value. All the functions, types, and macros in this section are defined in fenv.h.

C9X defines two data types, fenv_t and fexcept_t. The C9X draft standard does not define any details about these types, so for portable code you must treat them as opaque. The ARM compiler defines them to be structure types, for details see *ARM compiler extensions to the C9X interface* on page 6-20.

The type fenv_t is defined to hold all the information about the current floating-point environment:
*   the rounding mode
*   the exception sticky flags
*   whether each exception is masked
*   what handlers are installed, if any.

The type fexcept_t is defined to hold all the information relevant to a given set of exceptions.

C9X rounding mode and exception macros

C9X also defines a macro for each rounding mode and each exception. The macros are as follows:

```
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
FE_ALL_EXCEPT
FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD
```

The exception macros are bit fields. The macro FE_ALL_EXCEPT is the bitwise OR of all of them.

Handling exception flags

C9X provides three functions to clear, test and raise exceptions:

void feclearexcept(**int** excepts);**int** fetestexcept(**int** excepts);**void** feraiseexcept(**int** excepts);

The feclearexcept function clears the sticky flags for the given exceptions. The fetestexcept function returns the bitwise OR of the sticky flags for the given exceptions (so that if the Overflow flag was set but the Underflow flag was not, then calling fetestexcept(FE_OVERFLOW|FE_UNDERFLOW) would return FE_OVERFLOW).

The feraiseexcept function raises the given exceptions, in unspecified order. If an exception trap is enabled for an exception raised this way, it is called.

C9X also provides functions to save and restore everything about a given exception. This includes the sticky flag, whether the exception is trapped, and the address of the trap handler, if any. These functions are:

void fegetexceptflag(fexcept_t *flagp, **int** excepts);**void** fesetexceptflag(**const** fexcept_t *flagp, **int** excepts);

The fegetexceptflag function copies all the information relating to the given exceptions into the fexcept_t variable provided. The fesetexceptflag function copies all the information relating to the given exceptions from the fexcept_t variable into the current floating-point environment.

————— **Note** —————

fesetexceptflag can be used to set the sticky flag of a trapped exception to 1 without calling the trap handler, whereas feraiseexcept calls the trap handler for any trapped exception.

Handling rounding modes

C9X provides two functions for controlling rounding modes:

int fegetround(**void**);**int** fesetround(**int** round);

The fegetround function returns the current rounding mode, which has a value equal to that of one of the macros listed in *C9X rounding mode and exception macros* on page 6-17. The fesetround function sets the current rounding mode to the value provided. fesetround returns zero for success, or nonzero if its argument is not a valid rounding mode.

Saving the whole environment

C9X provides functions to save and restore the entire floating-point environment:

void fegetenv(fenv_t *envp);**void** fesetenv(**const** fenv_t *envp);

The fegetenv function stores the current state of the floating-point environment into the fenv_t variable provided. The fesetenv function restores the environment from the variable provided.

Like fesetexceptflag, fesetenv does not call trap handlers when it sets the sticky flags for trapped exceptions.

Temporarily disabling exceptions

C9X provides two functions that enable you to avoid risking exception traps when executing code that might cause exceptions. This is useful when, for example, trapped exceptions are using the ARM default behavior. The default is to cause **SIGFPE** and terminate the application.

int feholdexcept(fenv_t *envp);**void** feupdateenv(**const** fenv_t *envp);

The feholdexcept function saves the current floating-point environment in the fenv_t variable provided, sets all exceptions to be untrapped, and clears all the exception sticky flags. You can then execute code that might cause unwanted exceptions, and make sure the sticky flags for those exceptions are cleared. Then you can call feupdateenv. This restores any exception traps and calls them if necessary.

For example, suppose you have a function frob() that might cause the Underflow or Invalid Operation exceptions (assuming both exceptions are trapped). You are not interested in Underflow, but you want to know if an invalid operation is attempted. So you could do this:

```
fenv_t env;
feholdexcept(&env);
frob();
feclearexcept(FE_UNDERFLOW);
feupdateenv(&env);
```

Then, if the frob() function raises Underflow, it is cleared again by feclearexcept, and so no trap occurs when feupdateenv is called. However, if frob() raises Invalid Operation, the sticky flag is set when feupdateenv is called, and so the trap handler is invoked.

This mechanism is provided by C9X because C9X specifies no way to change exception trapping for individual exceptions. A better method is to use __ieee_status to disable the Underflow trap while leaving the Invalid Operation trap enabled. This has the

advantage that the Invalid Operation trap handler is provided with all the information about the invalid operation (that is, what operation was being performed, and on what data), and can invent a result for the operation. Using the C9X method, the Invalid Operation trap handler is called after the fact, receives no information about the cause of the exception, and is called too late to provide a substitute result.

6.3.5   ARM compiler extensions to the C9X interface

The ARM compiler provides some extensions to the C9X interface, to enable it to do everything that the ARM floating-point environment is capable of. This includes trapping and untrapping individual exception types, and also installing custom trap handlers.

The types fenv_t and fexcept_t are not defined by C9X to be anything in particular. The ARM compiler defines them both to be the same structure type:

```
typedef struct {
    unsigned statusword;
    __ieee_handler_t __invalid_handler;
    __ieee_handler_t __divbyzero_handler;
    __ieee_handler_t __overflow_handler;
    __ieee_handler_t __underflow_handler;
    __ieee_handler_t __inexact_handler;
} fenv_t, fexcept_t;
```

The members of this structure are:

- statusword is the same status variable that the function __ieee_status sees, laid out in the same format (see *__ieee_status()* on page 6-10).

- five function pointers giving the address of the trap handler for each exception. By default each is NULL. This means that if the exception is trapped then the default exception trap action happens. The default is to cause a **SIGFPE** signal.

Writing custom exception trap handlers

If you want to install a custom exception trap handler, declare it as a function like this:

```
__softfp__ieee_value_t myhandler(__ieee_value_t op1,
                                 __ieee_value_t op2,
                                 __ieee_edata_t edata);
```

The parameters to this function are:

- op1 and op2 are used to give the operands, or the intermediate result, for the operation that caused the exception:

    — For the Invalid Operation and Divide by Zero exceptions, the original operands are supplied.

    — For the Inexact Result exception, all that is supplied is the ordinary result that would have been returned anyway. This is provided in op1.

    — For the Overflow exception, an intermediate result is provided. This result is calculated by working out what the operation would have returned if the exponent range had been big enough, and then adjusting the exponent so that it fits in the format. The exponent is adjusted by 192 (0xC0) in single precision, and by 1536 (0x600) in double precision.

        If Overflow happens when converting a **double** to a **float**, the result is supplied in **double** format, rounded to single precision, with the exponent biased by 192.

    — For the Underflow exception, a similar intermediate result is produced, but the bias value is added to the exponent instead of being subtracted. The edata parameter also contains a flag to show whether the intermediate result has had to be rounded up, down, or not at all.

    The type __ieee_value_t is defined as a union of all the possible types that an operand can be passed as:

```
typedef union {
    float __f;
    float __s;
    double __d;
    int __i;
    unsigned int __ui;
#if !defined(__STRICT_ANSI__) || (defined(__STDC_VERSION__) && 199901L <=
__STDC_VERSION__)
    long long __l;
    unsigned long long __ul;
#endif                                  /* __STRICT_ANSI__ */
    struct { int __word1, __word2; } __str;
} __ieee_value_t;                       /* in/out values passed to traps */
```

——— **Note** ———

If you do not compile with --strict, and you have code that used the older definition of __ieee_value_t, your older code still works. See the file fenv.h for more details.

———————————

- edata contains flags that give details about the exception that occurred, and what operation was being performed. (The type `__ieee_edata_t` is a synonym for **unsigned int**.)

- The return value from the function is used as the result of the operation that caused the exception.

The flags contained in edata are:

- edata & FE_EX_RDIR is nonzero if the intermediate result in Underflow was rounded down, and 0 if it was rounded up or not rounded. (The difference between the last two is given in the Inexact Result bit.) This bit is meaningless for any other type of exception.

- edata & FE_EX_*exception* is nonzero if the given *exception* (INVALID, DIVBYZERO, OVERFLOW, UNDERFLOW, or INEXACT) occurred. This enables you to:

  — use the same handler function for more than one exception type (the function can test these bits to tell what exception it is supposed to handle)

  — determine whether Overflow and Underflow intermediate results have been rounded or are exact.

  Because the FE_EX_INEXACT bit can be set in combination with either FE_EX_OVERFLOW or FE_EX_UNDERFLOW, you must determine the type of exception that actually occurred by testing Overflow and Underflow before testing Inexact.

- edata & FE_EX_FLUSHZERO is nonzero if the FZ bit was set when the operation was performed (see *__ieee_status()* on page 6-10).

- edata & FE_EX_ROUND_MASK gives the rounding mode that applies to the operation. This is normally the same as the current rounding mode, unless the operation that caused the exception was a routine such as _ffix, that always rounds toward zero. The available rounding mode values are FE_EX_ROUND_NEAREST, FE_EX_ROUND_PLUSINF, FE_EX_ROUND_MINUSINF and FE_EX_ROUND_ZERO.

- edata & FE_EX_INTYPE_MASK gives the type of the operands to the function, as one of the type values shown in Table 6-9.

Table 6-9 FE_EX_INTYPE_MASK operand type flags

Flag	Operand type
FE_EX_INTYPE_FLOAT	**float**
FE_EX_INTYPE_DOUBLE	**double**
FE_EX_INTYPE_INT	**int**

Table 6-9 FE_EX_INTYPE_MASK operand type flags

Flag	Operand type
FE_EX_INTYPE_UINT	`unsigned int`
FE_EX_INTYPE_LONGLONG	`long long`
FE_EX_INTYPE_ULONGLONG	`unsigned long long`

- `edata & FE_EX_OUTTYPE_MASK` gives the type of the operands to the function, as one of the type values shown in Table 6-10.

Table 6-10 FE_EX_OUTTYPE_MASK operand type flags

Flag	Operand type
FE_EX_OUTTYPE_FLOAT	`float`
FE_EX_OUTTYPE_DOUBLE	`double`
FE_EX_OUTTYPE_INT	`int`
FE_EX_OUTTYPE_UINT	`unsigned int`
FE_EX_OUTTYPE_LONGLONG	`long long`
FE_EX_OUTTYPE_ULONGLONG	`unsigned long long`

- `edata & FE_EX_FN_MASK` gives the nature of the operation that caused the exception, as one of the operation codes shown in Table 6-11.

Table 6-11 FE_EX_FN_MASK operation type flags

Flag	Operation type
FE_EX_FN_ADD	Addition.
FE_EX_FN_SUB	Subtraction.
FE_EX_FN_MUL	Multiplication.
FE_EX_FN_DIV	Division.
FE_EX_FN_REM	Remainder.
FE_EX_FN_RND	Round to integer.
FE_EX_FN_SQRT	Square root.

Table 6-11 FE_EX_FN_MASK operation type flags

Flag	Operation type
FE_EX_FN_CMP	Compare.
FE_EX_FN_CVT	Convert between formats.
FE_EX_FN_LOGB	Exponent fetching.
FE_EX_FN_SCALBN	Scaling. ——— **Note** ——— The FE_EX_INTYPE_MASK flag only specifies the type of the first operand, the second operand is always an **int**.
FE_EX_FN_NEXTAFTER	Next representable. ——— **Note** ——— Both operands are the same type. Calls to nexttoward cause the value of the second operand to change to a value that is of the same type as the first operand. This does not affect the result.
FE_EX_FN_RAISE	The exception was raised explicitly, by feraiseexcept or feupdateenv. In this case almost nothing in the edata word is valid.

When the operation is a comparison, the result must be returned as if it were an **int**, and must be one of the four values shown in Table 6-12.

Input and output types are the same for all operations except Compare and Convert.

Table 6-12 FE_EX_CMPRET_MASK comparison type flags

Flag	Comparison
FE_EX_CMPRET_LESS	op1 is less than op2
FE_EX_CMPRET_EQUAL	op1 is equal to op2
FE_EX_CMPRET_GREATER	op1 is greater than op2
FE_EX_CMPRET_UNORDERED	op1 and op2 are not comparable

Example exception handler

Example 6-1 shows a custom exception handler. Suppose you are converting some Fortran code into C. The Fortran numerical standard requires 0 divided by 0 to be 1, whereas IEEE 754 defines 0 divided by 0 to be an Invalid Operation and so by default it returns a quiet NaN. The Fortran code is likely to rely on this behavior, and rather than modifying the code, it is probably easier to make 0 divided by 0 return 1.

When compiling, you must select a floating-point model that supports exceptions, for example --fpmode ieee_full or --fpmode ieee_fixed. See *Defining optimization criteria* on page 2-43 for more details.

After the handler is installed, dividing 0.0 by 0.0 returns 1.0.

Example 6-1

```
#include <fenv.h>
#include <signal.h>
#include <stdio.h>
__softfp __ieee_value_t myhandler(__ieee_value_t op1, __ieee_value_t op2,
                                  __ieee_edata_t edata)
{
    __ieee_value_t ret;
    if ((edata & FE_EX_FN_MASK) == FE_EX_FN_DIV) {
        if ((edata & FE_EX_INTYPE_MASK) == FE_EX_INTYPE_FLOAT) {
            if (op1.f == 0.0 && op2.f == 0.0) {
                ret.f = 1.0;
                return ret;
            }
        }
        if ((edata & FE_EX_INTYPE_MASK) == FE_EX_INTYPE_DOUBLE) {
            if (op1.d == 0.0 && op2.d == 0.0) {
                ret.d = 1.0;
                return ret;
            }
        }
    }
    /* For all other invalid operations, raise SIGFPE as usual */
    raise(SIGFPE);
}
int main(void)
{
    float i, j, k;
    fenv_t env;
    fegetenv(&env);
    env.statusword |= FE_IEEE_MASK_INVALID;
```

```
        env.invalid_handler = myhandler;
        fesetenv(&env);
        i = 0.0;
        j = 0.0;
        k = i/j;
        printf("k is %f\n", k);
}
```

Exception trap handling by signals

If an exception is trapped but the trap handler address is set to NULL, a default trap handler is used.

The default trap handler raises a **SIGFPE** signal. The default handler for **SIGFPE** prints an error message and terminates the program.

If you trap **SIGFPE**, you can declare your signal handler function to have a second parameter that tells you the type of floating-point exception that occurred. This feature is provided for compatibility with Microsoft products. The values are _FPE_INVALID, _FPE_ZERODIVIDE, _FPE_OVERFLOW, _FPE_UNDERFLOW and _FPE_INEXACT. They are defined in float.h. For example:

```
void sigfpe(int sig, int etype) {
    printf("SIGFPE (%s)\n",
            etype == _FPE_INVALID ? "Invalid Operation" :
            etype == _FPE_ZERODIVIDE ? "Divide by Zero" :
            etype == _FPE_OVERFLOW ? "Overflow" :
            etype == _FPE_UNDERFLOW ? "Underflow" :
            etype == _FPE_INEXACT ? "Inexact Result" :
            "Unknown");
}
signal(SIGFPE, (void(*)(int))sigfpe);
```

To generate your own **SIGFPE** signals with this extra information, you can call the function __rt_raise instead of the ISO function raise(). In Example 6-1 on page 6-25, instead of:

```
    raise(SIGFPE);
```

it is better to code:

```
    __rt_raise(SIGFPE, _FPE_INVALID);
```

__rt_raise is declared in rt_misc.h.

6.4 The math library, mathlib

In addition to the functions defined by the ISO C standard, mathlib provides the following C99 functions and macros. These are in addition to those provided by the floating-point library, fplib (see *The software floating-point library, fplib* on page 6-3):

mathlib also provides the following non-standard functions:

In RVCT v2.2, the C99 functionality provided by fplib and mathlib changes behavior exhibited by previous versions of these libraries. To aid porting there is a compatibility mode that emulates the previous behavior. This provides the following functions and macros:

- *Legacy, Exponent of a number (ilogb, ilogbf, ilogbl)* on page 6-34
- *Legacy, Determine if a number is finite (finite)* on page 6-34
- *Legacy, Determine if a number is a NaN (isnan)* on page 6-35.

——— **Note** ———

This legacy support will be removed in a future release such that only the new behavior remains.

6.4.1 Range reduction in mathlib

Trigonometric functions in mathlib use range reduction to bring large arguments within the range 0 to 2π. The ARM compiler provides two different range reduction functions. One is accurate to one unit in the last place for *any* input values, but is larger and slower than the other. The other is reliable enough for almost all purposes and is faster and smaller.

The fast and small range reducer is used by default. To select the more accurate one, use either:

- `#pragma import (__use_accurate_range_reduction)` from C
- `IMPORT __use_accurate_range_reduction` from assembly language.

6.4.2 Determine the type of a floating-point number (fpclassify)

This macro classifies its argument value based on type.

`int fpclassify(real-floating x);`

where *real-floating* can be type `float`, `double`, or `long double`.

Here, fpclassify returns one of the following values:

- `FP_INFINITE` if x is an infinity
- `FP_NAN` if x is a NaN
- `FP_NORMAL` if x is a normal
- `FP_SUBNORMAL` if x is a subnormal
- `FP_ZERO` if x is zero.

It does not cause any errors or exceptions.

6.4.3   Determine if a number is finite (isfinite)

This macro determines if its argument has a finite value.

int isfinite(*real-floating* x);

where *real-floating* can be type **float**, **double**, or **long double**.

Here, isfinite returns:
- non-zero if x is finite
- 0 if x is infinite or a NaN.

It does not cause any errors or exceptions.

6.4.4   Determine if a number is infinite (isinf)

This macro determines if its argument is an infinity.

int isinf(*real-floating* x);

where *real-floating* can be type **float**, **double**, or **long double**.

Here, isinf returns:
- non-zero if x is infinite
- 0 if x is finite or a NaN.

It does not cause any errors or exceptions.

6.4.5   Determine if a number is a NaN (isnan)

This macro determines if the value of its argument is a NaN.

int isnan(*real-floating* x);

where *real-floating* can be type **float**, **double**, or **long double**.

Here, isnan returns:
- non-zero if x is a NaN
- 0 otherwise.

It does not cause any errors or exceptions.

6.4.6   Determine if a number is normal (isnormal)

This macro determines if its argument value is normal.

int isnormal(*real-floating* x);

---

where *real-floating* can be type **float**, **double**, or **long double**.

Here, isnormal returns:
- non-zero if x is a normal
- 0 if x is infinite, a NaN, subnormal, or zero.

It does not cause any errors or exceptions.

6.4.7   Return the sign bit of a number (signbit)

This macro determines if the sign of its argument value is negative.

int signbit(*real-floating* x);

where *real-floating* can be type **float**, **double**, or **long double**.

Here, signbit returns:
- 1 if x is negative
- 0 if x is positive.

signbit returns a sign value for all values of x (including zeros and NaNs).

It does not cause any errors or exceptions.

6.4.8   Copy sign functions (copysign, copysignf)

These functions replace the sign bit of x with the sign bit of y.

double copysign(**double** x, **double** y);**float** copysignf(**float** x, **float** y);

These functions treat all floating-point numbers as signed (including zeros and NaNs) and do not cause any errors or exceptions.

6.4.9   Comparison macros (isgreater, isgreaterequal, isless, islessequal, islessgreater, isunordered)

These macros compare x and y as shown in Table 6-13 on page 6-31.

int isgreater(*real-floating* x, *real-floating* y);**int** isgreaterequal(*real-floating* x, *real-floating* y);**int** isless(*real-floating* x, *real-floating* y);**int** islessequal(*real-floating* x, *real-floating* y);**int** islessgreater(*real-floating* x, *real-floating* y);**int** isunordered(*real-floating* x, *real-floating* y);

where *real-floating* can be type **float**, **double**, or **long double**.

These macros do not cause any errors or exceptions, unlike the equivalent relationship operators that do raise exceptions on all NaNs.

Table 6-13 Comparison macros

Macro	Comparison
isgreater	x is greater than y
isgreaterequal	x is greater than or equal to y
isless	x is less than y
islessequal	x is less than or equal to y
islessgreater	x (is less than or greater than) y
isunordered	x is unordered with respect to y

6.4.10 Inverse hyperbolic functions (acosh, asinh, atanh)

These functions are the inverses of the ISO-required cosh, sinh and tanh.

double acosh(**double** x);**double** asinh(**double** x);**double** atanh(**double** x);

Here:

• acosh always has a choice of two return values, one positive and one negative, because cosh is a symmetric function (that is, it returns the same value when applied to *x* or –*x*). It chooses the positive result.

• acosh returns an EDOM error if called with an argument less than 1.0.

• atanh returns an EDOM error if called with an argument whose absolute value exceeds 1.0.

6.4.11 Cube root (cbrt)

This function returns the cube root of its argument.

double cbrt(**double** x);

6.4.12 Error functions (erf, erfc)

These functions compute the standard statistical error function, related to the Normal distribution:

double erf(**double** x);**double** erfc(**double** x);

Here:

- erf computes the ordinary error function of *x*

- erfc computes one minus erf(*x*). It is better to use erfc(*x*) than 1-erf(*x*) when *x* is large, because the answer is more accurate.

6.4.13 One less than exp(*x*) (expm1)

This function computes e^x, minus one. It is better to use expm1(x) than exp(x)-1 if *x* is very near to zero, because expm1 returns a more accurate value.

double expm1(**double** x);

6.4.14 Gamma function (gamma, gamma_r)

These functions both compute the logarithm of the gamma function. They are synonyms for lgamma and lgamma_r (see *The logarithm of the gamma function (lgamma, lgamma_r)* on page 6-33).

double gamma(**double** x);**double** gamma_r(**double** x, **int** *);

——— **Note** ———

Despite their names, these functions compute the logarithm of the gamma function, not the gamma function itself.

6.4.15 Hypotenuse function (hypot)

This function computes the length of the hypotenuse of a right-angled triangle whose other two sides have length *x* and *y*. Equivalently, it computes the length of the vector (*x*,*y*) in Cartesian coordinates. Using hypot(x,y) is better than sqrt(x*x+y*y) because some values of *x* and *y* might cause x * x + y * y to overflow even though its square root would not.

double hypot(**double** x, **double** y);

hypot returns an ERANGE error when the result does not fit in a **double**.

6.4.16 Bessel functions of the first kind (j0, j1, jn)

These functions compute Bessel functions of the first kind. j0 and j1 compute the functions of order 0 and 1 respectively. j*n* computes the function of order *n*.

double j0(**double** x);**double** j1(**double** x);**double** jn(**int** n, **double** x);

If the absolute value of x exceeds π times 2^{52}, these functions return an ERANGE error, denoting total loss of significance in the result.

6.4.17 The logarithm of the gamma function (lgamma, lgamma_r)

These functions compute the logarithm of the absolute value of the gamma function of x. The sign of the function is returned separately, so that the two can be used to compute the actual gamma function of x.

double lgamma(**double** x);**double** lgamma_r(**double** x, **int** *sign);

lgamma returns the sign of the gamma function of x in the global variable signgam. lgamma_r returns it in a user variable, whose address is passed in the sign parameter. The value, in either case, is either +1 or –1.

Both functions return an ERANGE error if the answer is too big to fit in a **double**.

Both functions return an EDOM error if x is zero or a negative integer.

6.4.18 Logarithm of one more than x (log1p)

This function computes the natural logarithm of $x + 1$. Like expm1, it is better to use this function than log(x+1) because this function is more accurate when x is near zero.

double log1p(**double** x);

6.4.19 IEEE 754 remainder function (remainder)

This function is the IEEE 754 remainder operation. It is a synonym for _drem (see *Arithmetic on numbers in a particular format* on page 6-4).

double remainder(**double** x, **double** y);

6.4.20 IEEE round-to-integer operation (rint)

This function is the IEEE 754 round-to-integer operation. It is a synonym for _drnd (see *Arithmetic on numbers in a particular format* on page 6-4).

double rint(**double** x);

6.4.21 Return the fraction part of a number (significand)

This function returns the fraction part of *x*, as a number between 1.0 and 2.0 (not including 2.0).

double significand(**double** x);

6.4.22 Bessel functions of the second kind (y0, y1, yn)

These functions compute Bessel functions of the second kind. y0 and y1 compute the functions of order 0 and 1 respectively. yn computes the function of order *n*.

double y0(**double** x);**double** y1(**double** x);**double** yn(**int**, **double**);

If *x* is positive and exceeds π times 2^{52}, these functions return an ERANGE error, denoting total loss of significance in the result.

6.4.23 Legacy, Exponent of a number (ilogb, ilogbf, ilogbl)

The values returned by ilogb, ilogbf, and ilogbl for input NaNs have changed in the current release of RVCT. This is different from the behavior in RVCT v2.1 and earlier.

To support legacy code that requires the old return values for NaNs, you can enable a legacy mode by defining the macro __ENABLE_LEGACY_MATHLIB before including math.h. This defines three macros (ilogb, ilogbf, and ilogbl) to overlay the standard functions and so return the old values. However, be aware that taking the address of the ilogb functions still returns the new behavior.

─────── **Note** ───────

This legacy support will be removed in a future release such that only the new behavior remains. ARM recommends that you migrate your use of these macros to the new functions.

─────────────────────

6.4.24 Legacy, Determine if a number is finite (finite)

In RVCT v2.1 and earlier, mathlib included a finite function that provided similar functionality to the isfinite macro in the current release (see *Determine if a number is finite (isfinite)* on page 6-29).

To support legacy code that requires the `finite` function, you can enable a legacy mode by defining the macro `__ENABLE_LEGACY_MATHLIB` before including `math.h`. This defines a macro (`finite`) to provide the same behavior as the old `finite` function. However, be aware that it is no longer possible to take the address of the `finite` function and so you must change any code that does this.

———— **Note** ————

This legacy support will be removed in a future release such that only the new behavior remains. ARM recommends that you migrate your use of this function to the new macro.

6.4.25 Legacy, Determine if a number is a NaN (isnan)

In the current release of RVCT, mathlib provides `isnan` as a macro (see *Determine if a number is a NaN (isnan)* on page 6-29). In RVCT v2.1 and earlier, mathlib provided `isnan` as a function that only took **double** inputs (and was not able to handle **float**s).

To support legacy code that requires the `isnan` function, you can enable a legacy mode by defining the macro `__ENABLE_LEGACY_MATHLIB` before including `math.h`. This changes the definition of the `isnan` macro so that it behaves in the same way as the function provided with RVCT v2.1 and earlier. However, be aware that it is no longer possible to take the address of the `isnan` function and so you must change any code that does this.

———— **Note** ————

This legacy support will be removed in a future release such that only the new behavior remains. ARM recommends that you migrate your use of this function to the new macro.

6.5   IEEE 754 arithmetic

The ARM floating-point environment is an implementation of the IEEE 754 standard for binary floating-point arithmetic. This section contains a summary of the standard as it is implemented by the ARM compiler.

This section includes:

*   *Basic data types*
*   *Arithmetic and rounding* on page 6-39
*   *Exceptions* on page 6-40.

6.5.1   Basic data types

ARM floating-point values are stored in one of two data types, *single precision* and *double precision*. In this document these are called **float** and **double**. These are the corresponding C types.

Single precision

A **float** value is 32 bits wide. The structure is shown in Figure 6-3.

Figure 6-3 IEEE 754 single-precision floating-point format

The S field gives the sign of the number. It is 0 for positive, or 1 for negative.

The Exp field gives the exponent of the number, as a power of two. It is *biased* by 0x7F (127), so that very small numbers have exponents near zero and very large numbers have exponents near 0xFF (255).

So, for example:

*   if *Exp* = 0x7D (125), the number is between 0.25 and 0.5 (not including 0.5)
*   if *Exp* = 0x7E (126), the number is between 0.5 and 1.0 (not including 1.0)
*   if *Exp* = 0x7F (127), the number is between 1.0 and 2.0 (not including 2.0)
*   if *Exp* = 0x80 (128), the number is between 2.0 and 4.0 (not including 4.0)
*   if *Exp* = 0x81 (129), the number is between 4.0 and 8.0 (not including 8.0).

The Frac field gives the fractional part of the number. It usually has an implicit 1 bit on the front that is not stored to save space.

So if *Exp* is 0x7F, for example:

- if *Frac* = 00000000000000000000000 (binary), the number is 1.0
- if *Frac* = 10000000000000000000000 (binary), the number is 1.5
- if *Frac* = 01000000000000000000000 (binary), the number is 1.25
- if *Frac* = 11000000000000000000000 (binary), the number is 1.75.

So in general, the numeric value of a bit pattern in this format is given by the formula:

$$(-1)^S * 2^{Exp(-0x7F)} * (1 + Frac * 2^{-23})$$

Numbers stored in this form are called *normalized* numbers.

The maximum and minimum exponent values, 0 and 255, are special cases. Exponent 255 is used to represent infinity, and store *Not a Number* (NaN) values. Infinity can occur as a result of dividing by zero, or as a result of computing a value that is too large to store in this format. NaN values are used for special purposes. Infinity is stored by setting Exp to 255 and Frac to all zeros. If Exp is 255 and Frac is nonzero, the bit pattern represents a NaN.

Exponent 0 is used to represent very small numbers in a special way. If *Exp* is zero, then the Frac field has no implicit 1 on the front. This means that the format can store 0.0, by setting both Exp and Frac to all 0 bits. It also means that numbers that are too small to store using Exp >= 1 are stored with less precision than the ordinary 23 bits. These are called *denormals*.

Double precision

A **double** value is 64 bits wide. Figure 6-4 shows its structure.

63 62	52 51	0
S	Exp	Frac
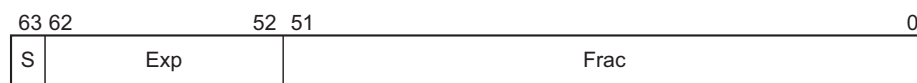
Figure 6-4 IEEE 754 double-precision floating-point format

As before, S is the sign, Exp the exponent, and Frac the fraction. Most of the discussion of **float** values remains true, except that:

- The Exp field is biased by 0x3FF (1023) instead of 0x7F, so numbers between 1.0 and 2.0 have an Exp field of 0x3FF.

- The Exp value used to represent infinity and NaNs is 0x7FF (2047) instead of 0xFF.

Sample values

Some sample **float** and **double** bit patterns, together with their mathematical values, are given in Table 6-14 and Table 6-15 on page 6-39.

Table 6-14 Sample single-precision floating-point values

Float value	S	Exp	Frac	Mathematical value	Notes
0x3F800000	0	0x7F	000...000	1.0	-
0xBF800000	1	0x7F	000...000	–1.0	-
0x3F800001	0	0x7F	000...001	1.000 000 119	a
0x3F400000	0	0x7E	100...000	0.75	-
0x00800000	0	0x01	000...000	$1.18*10^{-38}$	b
0x00000001	0	0x00	000...001	$1.40*10^{-45}$	c
0x7F7FFFFF	0	0xFE	111...111	$3.40*10^{38}$	d
0x7F800000	0	0xFF	000...000	Plus infinity	-
0xFF800000	1	0xFF	000...000	Minus infinity	-
0x00000000	0	0x00	000...000	0.0	e
0x7F800001	0	0xFF	000...001	Signaling NaN	f
0x7FC00000	0	0xFF	100...000	Quiet NaN	f

a. The smallest representable number that can be seen to be greater than 1.0. The amount that it differs from 1.0 is known as the *machine epsilon*. This is 0.000 000 119 in **float**, and 0.000 000 000 000 000 222 in **double**. The machine epsilon gives a rough idea of the number of significant figures the format can keep track of. **float** can do six or seven places. **double** can do fifteen or sixteen.

b. The smallest value that can be represented as a normalized number in each format. Numbers smaller than this can be stored as denormals, but are not held with as much precision.

c. The smallest positive number that can be distinguished from zero. This is the absolute lower limit of the format.

d. The largest finite number that can be stored. Attempting to increase this number by addition or multiplication causes overflow and generates infinity (in general).

e. Zero. Strictly speaking, they show plus zero. Zero with a sign bit of 1, minus zero, is treated differently by some operations, although the comparison operations (for example == and !=) report that the two types of zero are equal.

f. There are two types of NaNs, signaling NaNs and quiet NaNs. Quiet NaNs have a 1 in the first bit of Frac, and signaling NaNs have a zero there. The difference is that signaling NaNs cause an exception (see *Exceptions* on page 6-40) when used, whereas quiet NaNs do not.

Table 6-15 Sample double-precision floating-point values

Double value	S	Exp	Frac	Mathematical value	Notes
0x3FF00000 00000000	0	0x3FF	000...000	1.0	-
0xBFF00000 00000000	1	0x3FF	000...000	–1.0	-
0x3FF00000 00000001	0	0x3FF	000...001	1.000 000 000 000 000 222	[a]
0x3FE80000 00000000	0	0x3FE	100...000	0.75	-
0x00100000 00000000	0	0x001	000...000	$2.23*10^{-308}$	[b]
0x00000000 00000001	0	0x000	000...001	$4.94*10^{-324}$	[c]
0x7FEFFFFF FFFFFFFF	0	0x7FE	111...111	$1.80*10^{308}$	[d]
0x7FF00000 00000000	0	0x7FF	000...000	Plus infinity	-
0xFFF00000 00000000	1	0x7FF	000...000	Minus infinity	-
0x00000000 00000000	0	0x000	000...000	0.0	[e]
0x7FF00000 00000001	0	0x7FF	000...001	Signaling NaN	[f]
0x7FF80000 00000000	0	0x7FF	100...000	Quiet NaN	[f]

a. to f. For footnotes, see Table 6-14 on page 6-38.

6.5.2 Arithmetic and rounding

Arithmetic is generally performed by computing the result of an operation as if it were stored exactly (to infinite precision), and then rounding it to fit in the format. Apart from operations whose result already fits exactly into the format (such as adding 1.0 to 1.0), the correct answer is generally somewhere between two representable numbers in the format. The system then chooses one of these two numbers as the rounded result. It uses one of the following methods:

Round to nearest

The system chooses the nearer of the two possible outputs. If the correct answer is exactly half-way between the two, the system chooses the one where the least significant bit of Frac is zero. This behavior (round-to-even) prevents various undesirable effects.

This is the default mode when an application starts up. It is the only mode supported by the ordinary floating-point libraries. The enhanced floating-point libraries support all four rounding modes. See *Library naming conventions* on page 5-128.)

Round up, or round toward plus infinity

> The system chooses the larger of the two possible outputs (that is, the one further from zero if they are positive, and the one closer to zero if they are negative).

Round down, or round toward minus infinity

> The system chooses the smaller of the two possible outputs (that is, the one closer to zero if they are positive, and the one further from zero if they are negative).

Round toward zero, or chop, or truncate

> The system chooses the output that is closer to zero, in all cases.

6.5.3   Exceptions

Floating-point arithmetic operations can run into various problems. For example, the result computed might be either too big or too small to fit into the format, or there might be no way to calculate the result (as in trying to take the square root of a negative number, or trying to divide zero by zero). These are known as exceptions, because they indicate unusual or exceptional situations.

The ARM floating-point environment can handle exceptions in more than one way.

Ignoring exceptions

The system invents a plausible result for the operation and returns that. For example, the square root of a negative number can produce a NaN, and trying to compute a value too big to fit in the format can produce infinity. If an exception occurs and is ignored, a flag is set in the floating-point status word to tell you that something went wrong at some point in the past.

Trapping exceptions

This means that when an exception occurs, a piece of code called a trap handler is run. The system provides a default trap handler, that prints an error message and terminates the application. However, you can supply your own trap handlers, that can clean up the exceptional condition in whatever way you choose. Trap handlers can even supply a result to be returned from the operation.

For example, if you had an algorithm where it was convenient to assume that 0 divided by 0 was 1, you could supply a custom trap handler for the Invalid Operation exception, that spotted that particular case and substituted the answer you wanted.

Types of exception

The ARM floating-point environment recognizes the following types of exception:

- The Invalid Operation exception happens when there is no sensible result for an operation. This can happen for any of the following reasons:
  — performing any operation on a signaling NaN, except the simplest operations (copying and changing the sign)
  — adding plus infinity to minus infinity, or subtracting an infinity from itself
  — multiplying infinity by zero
  — dividing 0 by 0, or dividing infinity by infinity
  — taking the remainder from dividing anything by 0, or infinity by anything
  — taking the square root of a negative number (not including minus zero)
  — converting a floating-point number to an integer if the result does not fit
  — comparing two numbers if one of them is a NaN.

  If the Invalid Operation exception is not trapped, all these operations return a quiet NaN, except for conversion to an integer, that returns zero (as there are no quiet NaNs in integers).

- The Divide by Zero exception happens if you divide a finite nonzero number by zero. (Dividing zero by zero gives an Invalid Operation exception. Dividing infinity by zero is valid and returns infinity.)

  If Divide by Zero is not trapped, the operation returns infinity.

- The Overflow exception happens when the result of an operation is too big to fit into the format. This happens, for example, if you add the largest representable number (marked d in Table 6-14 on page 6-38) to itself.

  If Overflow is not trapped, the operation returns infinity, or the largest finite number, depending on the rounding mode.

- The Underflow exception can happen when the result of an operation is too small to be represented as a normalized number (with Exp at least 1).

  The situations that cause Underflow depend on whether it is trapped or not:
  — If Underflow is trapped, it occurs whenever a result is too small to be represented as a normalized number.
  — If Underflow is not trapped, it only occurs if the result actually loses accuracy because it is so small. So, for example, dividing the **float** number 0x00800000 by 2 does not signal Underflow, because the result (0x00400000) is still as accurate as it would be if *Exp* had a greater range. However, trying to multiply the **float** number 0x00000001 by 1.5 does signal Underflow.

(For readers familiar with the IEEE 754 specification, the chosen implementation options in the ARM compiler are to detect tininess after rounding, and to detect loss of accuracy as a denormalization loss.)

If Underflow is not trapped, the result is rounded to one of the two nearest representable denormal numbers, according to the current rounding mode. The loss of precision is ignored and the system returns the best result it can.

—   The Inexact Result exception happens whenever the result of an operation requires rounding. This would cause significant loss of speed if it had to be detected on every operation in software, so the ordinary floating-point libraries do not support the Inexact Result exception. The enhanced floating-point libraries all support Inexact Result.

If Inexact Result is not trapped, the system rounds the result in the usual way.

The flag for Inexact Result is also set by Overflow and Underflow if either one of those is not trapped.

All exceptions are untrapped by default.

Chapter 7
Semihosting

This chapter describes the semihosting mechanism. It contains the following sections:

- *Semihosting* on page 7-2
- *Semihosting implementation* on page 7-5
- *Semihosting SWIs* on page 7-6
- *Debug agent interaction SWIs* on page 7-20.

7.1 Semihosting

Semihosting enables code running on an ARM® target to use the I/O facilities on a host computer that is running an ARM debugger, for example RealView® Debugger or AXD. Examples of these facilities include keyboard input, screen output, and disk I/O.

This section describes the semihosting mechanism:

- *What is semihosting?*
- *The SWI interface* on page 7-3.

7.1.1 What is semihosting?

Semihosting is a mechanism for ARM targets to communicate input/output requests from application code to a host computer running a debugger. This mechanism could be used, for example, to enable functions in the C library, such as printf() and scanf(), to use the screen and keyboard of the host rather than having a screen and keyboard on the target system.

This is useful because development hardware often does not have all the input and output facilities of the final system. Semihosting enables the host computer to provide these facilities.

The semihosting SWI interface is common across all debug agents provided by ARM Limited. Semihosted operations work when you are using RVI-ME without any requirement for porting.

Semihosting is implemented by a set of defined Software Interrupt (SWI) operations. The application invokes the appropriate SWI and the debug agent then handles the SWI exception. The debug agent provides the required communication with the host.

Figure 7-1 on page 7-3 shows an overview of semihosting.

In many cases, the semihosting SWI is invoked by code within library functions. The application can also invoke the semihosting SWI directly. See the C library descriptions in Chapter 5 *The C and C++ Libraries* for more information on support for semihosting in the ARM C library.
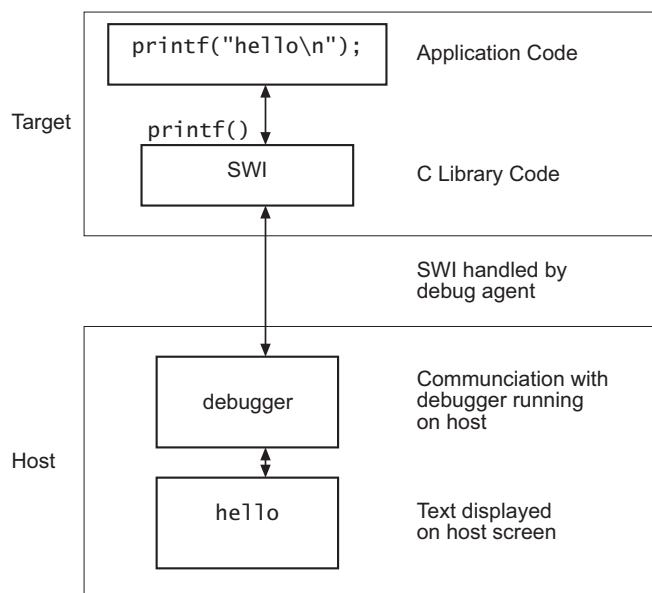
Figure 7-1 Semihosting overview

7.1.2 The SWI interface

The ARM and Thumb® SWI instructions contain a field that encodes the SWI number used by the application code. This number can be decoded by the SWI handler in the system.

Semihosting operations are requested using a single SWI number. This leaves the other SWI numbers available for use by the application or operating system. The SWI used for semihosting is:

0x123456   in ARM state

0xAB   in Thumb state.

The SWI number indicates to the debug agent that the SWI is a semihosting request. To distinguish between operations, the operation type is passed in r0. All other parameters are passed in a block that is pointed to by r1.

The result is returned in r0, either as an explicit return value or as a pointer to a data block. Even if no result is returned, assume that r0 is corrupted.

The available semihosting operation numbers passed in r0 are allocated as follows:

0x00-0x31   These are used by ARM.

0x32-0xFF   These are reserved for future use by ARM.

0x100-0x1FF   These are reserved for user applications. They are not used by ARM.

              If you are writing your own SWI operations, however, you are advised to use a different SWI number rather than using the semihosted SWI number and these operation type numbers.

0x200-0xFFFFFFFF

              These are undefined and are currently unused. It is recommended that you do not use these.

In the following sections, the number in parentheses after the operation name is the value placed into r0, for example SYS_OPEN (0x01).

If you are calling SWIs from assembly language code ARM recommends that you use the operation names defined in semihost.h. You can define the operation names with an EQU directive. For example:

```
SYS_OPEN    EQU 0x01
SYS_CLOSE   EQU 0x02
```

Changing the semihosting SWI numbers

It is strongly recommended that you do not change the semihosting SWI numbers 0x123456 (ARM) or 0xAB (Thumb). If you do, you must:

• change all the code in your system, including library code, to use the new SWI number

• reconfigure your debugger to use the new SWI number.

7.2   Semihosting implementation

The functionality provided by semihosting is basically the same on all debug agents. However, the implementation of semihosting differs between hosts.

This section describes the semihosting implementation on different debug agents:
- *RealView ICE Micro Edition*
- *RealMonitor*

7.2.1   RealView ICE Micro Edition

When using the RVI-ME DLL, semihosting is handled with either a real SWI exception handler, or by emulating a handler using breakpoints.

7.2.2   RealMonitor

RealMonitor implements a SWI handler that must be integrated with your system to enable semihosting support.

When the target executes a semihosted SWI instruction, the RealMonitor SWI handler carries out the required communication with the host.

For more information see the documentation supplied with RealMonitor.

7.3 Semihosting SWIs

The SWIs listed in Table 7-1 implement the semihosted operations. These operations are used by C library functions such as printf() and scanf(). They can be treated as AAPCS function calls. However, except for r0 that contains the return status, they restore the registers they are called with before returning.

Table 7-1 Semihosting SWIs

SWI	Description
SYS_OPEN (0x01) on page 7-7	Open a file on the host
SYS_CLOSE (0x02) on page 7-8	Close a file on the host
SYS_WRITEC (0x03) on page 7-8	Write a character to the console
SYS_WRITE0 (0x04) on page 7-8	Write a null-terminated string to the console
SYS_WRITE (0x05) on page 7-9	Write to a file on the host
SYS_READ (0x06) on page 7-9	Read the contents of a file into a buffer
SYS_READC (0x07) on page 7-10	Read a byte from the console
SYS_ISERROR (0x08) on page 7-10	Determine if a return code is an error
SYS_ISTTY (0x09) on page 7-11	Check whether a file is connected to an interactive device
SYS_SEEK (0x0A) on page 7-11	Seek to a position in a file
SYS_FLEN (0x0C) on page 7-12	Return the length of a file
SYS_TMPNAM (0x0D) on page 7-13	Return a temporary name for a file
SYS_REMOVE (0x0E) on page 7-14	Remove a file from the host
SYS_RENAME (0x0F) on page 7-14	Rename a file on the host
SYS_CLOCK (0x10) on page 7-15	Number of centiseconds since execution started
SYS_TIME (0x11) on page 7-15	Number of seconds since January 1, 1970
SYS_SYSTEM (0x12) on page 7-16	Pass a command to the host command-line interpreter
SYS_ERRNO (0x13) on page 7-16	Get the value of the C library errno variable
SYS_GET_CMDLINE (0x15) on page 7-17	Get the command-line used to call the executable

SWI	Description
SYS_HEAPINFO (0x16) on page 7-18	Get the system heap parameters
SYS_ELAPSED (0x30) on page 7-19	Get the number of target ticks since execution started
SYS_TICKFREQ (0x31) on page 7-19	Determine the tick frequency

7.3.1   SYS_OPEN (0x01)

Opens a file on the host system. The file path is specified either as relative to the current directory of the host process, or absolute, using the path conventions of the host operating system.

ARM targets interpret the special path name :tt as meaning the console input stream (for an open-read) or the console output stream (for an open-write). Opening these streams is performed as part of the standard startup code for those applications that reference the C stdio streams.

Entry

On entry, r1 contains a pointer to a three-word argument block:

word 1   This is a pointer to a null-terminated string containing a file or device name.

word 2   This is an integer that specifies the file opening mode. Table 7-2 gives the valid values for the integer, and their corresponding ISO C fopen() mode.

word 3   This is an integer that gives the length of the string pointed to by word 1.

The length does not include the terminating null character that must be present.

Table 7-2 Value of mode

mode	0	1	2	3	4	5	6	7	8	9	10	11
ISO C fopen mode[a]	r	rb	r+	r+b	w	wb	w+	w+b	a	ab	a+	a+b

a.   The non-ANSI option t is not supported.

Return

On exit, `r0` contains:
- a nonzero handle if the call is successful
- −1 if the call is not successful.

7.3.2   SYS_CLOSE (0x02)

Closes a file on the host system. The handle must reference a file that was opened with `SYS_OPEN`.

Entry

On entry, `r1` contains a pointer to a one-word argument block:

word 1   This is a file handle referring to an open file.

Return

On exit, `r0` contains:
- 0 if the call is successful
- −1 if the call is not successful.

7.3.3   SYS_WRITEC (0x03)

Writes a character byte, pointed to by `r1`, to the debug channel. When executed under an ARM debugger, the character appears on the host debugger console.

Entry

On entry, `r1` contains a pointer to the character.

Return

None. Register `r0` is corrupted.

7.3.4   SYS_WRITE0 (0x04)

Writes a null-terminated string to the debug channel. When executed under an ARM debugger, the characters appear on the host debugger console.

Entry

On entry, r1 contains a pointer to the first byte of the string.

Return

None. Register r0 is corrupted.

7.3.5   SYS_WRITE (0x05)

Writes the contents of a buffer to a specified file at the current file position. The file position is specified either:
- explicitly, by a SYS_SEEK
- implicitly as one byte beyond the previous SYS_READ or SYS_WRITE request.

The file position is at the start of the file when the file is opened, and is lost when the file is closed.

Perform the file operation as a single action whenever possible. For example, do not split a write of 16KB into four 4KB chunks unless there is no alternative.

Entry

On entry, r1 contains a pointer to a three-word data block:

word 1   This contains a handle for a file previously opened with SYS_OPEN.

word 2   This points to the memory containing the data to be written.

word 3   This contains the number of bytes to be written from the buffer to the file.

Return

On exit, r0 contains:
- 0 if the call is successful
- the number of bytes that are not written, if there is an error.

7.3.6   SYS_READ (0x06)

Reads the contents of a file into a buffer. The file position is specified either:
- explicitly by a SYS_SEEK
- implicitly one byte beyond the previous SYS_READ or SYS_WRITE request.

The file position is at the start of the file when the file is opened, and is lost when the file is closed. Perform the file operation as a single action whenever possible. For example, do not split a read of 16KB into four 4KB chunks unless there is no alternative.

Entry

On entry, `r1` contains a pointer to a four-word data block:

word 1   This contains a handle for a file previously opened with `SYS_OPEN`.

word 2   This points to a buffer.

word 3   This contains the number of bytes to read to the buffer from the file.

Return

On exit:

*   `r0` contains zero if the call is successful

*   If `r0` contains the same value as word 3, the call has failed and EOF is assumed

*   If `r0` contains a greater value than word 3, the call was partially successful. No
    error is assumed, but the buffer has not been filled.

If the handle is for an interactive device (that is, `SYS_ISTTY` returns –1 for this handle), a
nonzero return from `SYS_READ` indicates that the line read did not fill the buffer.

7.3.7   SYS_READC (0x07)

Reads a byte from the console.

Entry

Register `r1` must contain zero. There are no other parameters or values possible.

Return

On exit, `r0` contains the byte read from the console.

7.3.8   SYS_ISERROR (0x08)

Determines whether the return code from another semihosting call is an error status or
not. This call is passed a parameter block containing the error code to examine.

Entry

On entry, `r1` contains a pointer to a one-word data block:

word 1   This is the required status word to check.

Return

On exit, r0 contains:

- 0 if the status word is not an error indication
- a nonzero value if the status word is an error indication.

7.3.9   SYS_ISTTY (0x09)

Checks whether a file is connected to an interactive device.

Entry

On entry, r1 contains a pointer to a one-word argument block:

word 1   This is a handle for a previously opened file object.

Return

On exit, r0 contains:

- 1 if the handle identifies an interactive device
- 0 if the handle identifies a file
- a value other than 1 or 0 if an error occurs.

7.3.10   SYS_SEEK (0x0A)

Seeks to a specified position in a file using an offset specified from the start of the file. The file is assumed to be a byte array and the offset is given in bytes.

Entry

On entry, r1 contains a pointer to a two-word data block:

word 1   This is a handle for a seekable file object.

word 2   The absolute byte position to search to.

Return

On exit, r0 contains:

- 0 if the request is successful

- A negative value if the request is not successful. SYS_ERRNO can be used to read the value of the host errno variable describing the error.

——— **Note** ———

The effect of seeking outside the current extent of the file object is undefined.

7.3.11 SYS_FLEN (0x0C)

Returns the length of a specified file.

Entry

On entry, r1 contains a pointer to a one-word argument block:

word 1   This is a handle for a previously opened, seekable file object.

Return

On exit, r0 contains:
- the current length of the file object, if the call is successful
- –1 if an error occurs.

7.3.12 SYS_TMPNAM (0x0D)

Returns a temporary name for a file identified by a system file identifier.

Entry

On entry, r1 contains a pointer to a three-word argument block:

word 1   This is a pointer to a buffer.

word 2   This is a target identifier for this filename. Its value must be an integer in the range 0 to 255.

word 3   This contains the length of the buffer. The length must be at least the value of L_tmpnam on the host system.

Return

On exit, r0 contains:
- 0 if the call is successful
- –1 if an error occurs.

The buffer pointed to by r1 contains the filename, prefixed with a suitable directory name.

If you use the same target identifier again, the same filename is returned.

7.3.13   SYS_REMOVE (0x0E)

--------- **Caution** ---------

Deletes a specified file on the host filing system.

Entry

On entry, r1 contains a pointer to a two-word argument block:

word 1   This points to a null-terminated string that gives the path name of the file
to be deleted.

word 2   This is the length of the string.

Return

On exit, r0 contains:
*   0 if the delete is successful
*   a nonzero, host-specific error code if the delete fails.

7.3.14   SYS_RENAME (0x0F)

Renames a specified file.

Entry

On entry, r1 contains a pointer to a four-word data block:
word 1   This is a pointer to the name of the old file.
word 2   This is the length of the old filename.
word 3   This is a pointer to the new filename.
word 4   This is the length of the new filename.

Both strings are null-terminated.

Return

On exit, r0 contains:
*   0 if the rename is successful
*   a nonzero, host-specific error code if the rename fails.

7.3.15   SYS_CLOCK (0x10)

Returns the number of centiseconds since the execution started.

Values returned by this SWI can be of limited use for some benchmarking purposes because of communication overhead or other agent-specific factors. For example, with RVI-ME the request is passed back to the host for execution. This can lead to unpredictable delays in transmission and process scheduling.

Use this function to calculate time intervals (the length of time some action took) by calculating differences between intervals with and without the code sequence to be timed

Some systems enable more accurate timing (see *SYS_ELAPSED (0x30)* on page 7-19 and *SYS_TICKFREQ (0x31)* on page 7-19).

Entry

Register r1 must contain zero. There are no other parameters.

Return

On exit, r0 contains:

*   the number of centiseconds since some arbitrary start point, if the call is successful

*   –1 if the call is unsuccessful (for example, because of a communications error).

7.3.16   SYS_TIME (0x11)

Returns the number of seconds since 00:00 January 1, 1970. This is real-world time, regardless of any RVI-ME configuration.

Entry

There are no parameters.

Return

On exit, r0 contains the number of seconds.

7.3.17   SYS_SYSTEM (0x12)

Passes a command to the host command-line interpreter. This enables you to execute a system command such as `dir`, `ls`, or `pwd`. The terminal I/O is on the host, and is not visible to the target.

——— **Caution** ———

The command passed to the host is executed on the host. Ensure that any command passed has no unintended consequences.

Entry

On entry, `r1` contains a pointer to a two-word argument block:

word 1   This points to a string that is to be passed to the host command-line interpreter.

word 2   This is the length of the string.

Return

On exit, `r0` contains the return status.

7.3.18   SYS_ERRNO (0x13)

Returns the value of the C library `errno` variable associated with the host implementation of the semihosting SWIs. The `errno` variable can be set by a number of C library semihosted functions, including:

- SYS_REMOVE
- SYS_OPEN
- SYS_CLOSE
- SYS_READ
- SYS_WRITE
- SYS_SEEK.

Whether `errno` is set or not, and to what value, is entirely host-specific, except where the ISO C standard defines the behavior.

Entry

There are no parameters. Register `r1` must be zero.

Return

On exit, r0 contains the value of the C library errno variable.

7.3.19   SYS_GET_CMDLINE (0x15)

Returns the command line used to call the executable (that is, argc and argv).

Entry

On entry, r1 points to a two-word data block to be used for returning the command string and its length:

word 1   This is a pointer to a buffer of at least the size specified in word two.

word 2   This is the length of the buffer in bytes.

Return

On exit:

*   Register r1 points to a two-word data block:

    word 1   This is a pointer to null-terminated string of the command line.

    word 2   This is the length of the string.

    The debug agent might impose limits on the maximum length of the string that can be transferred. However, the agent must be able to transfer a command line of at least 80 bytes.

*   Register r0 contains an error code:

    —   0 if the call is successful

    —   –1 if the call is unsuccessful (for example, because of a communications error).

7.3.20 SYS_HEAPINFO (0x16)

Returns the system stack and heap parameters. The values returned are typically those used by the C library during initialization. For RVI-ME, the values returned are the image location and the top of memory.

The C library can override these values (see *Tailoring storage management* on page 5-77 for more information on memory management in the C library).

The host debugger determines the actual values to return by using the top_of_memory debugger variable.

Entry

On entry, r1 contains the address of a pointer to a four-word data block. The contents of the data block are filled by the function. See Example 7-1 for the structure of the data block and return values.

Example 7-1

```
struct block {
    int heap_base;
    int heap_limit;
    int stack_base;
    int stack_limit;
};
struct block *mem_block, info;
mem_block = &info;
AngelSWI(SYS_HEAPINFO, (unsigned) &mem_block);
```

——— **Note** ———

If word one of the data block has the value zero, the C library replaces the zero with Image$$ZI$$Limit. This value corresponds to the top of the data region in the memory map.

Return

On exit, r1 contains the address of the pointer to the structure.

If one of the values in the structure is 0, the system was unable to calculate the real value.

7.3.21   SYS_ELAPSED (0x30)

Returns the number of elapsed target ticks since execution started. Use SYS_TICKFREQ to determine the tick frequency.

Entry

On entry, r1 points to a two-word data block to be used for returning the number of elapsed ticks:

word 1   The least significant word in the doubleword value.

word 2   The most significant word.

Return

On exit:

*   r0 contains –1 if r1 does point to a doubleword containing the number of elapsed ticks. RVI-ME does not support this SWI and always returns –1 in r0.

*   r1 points to a doubleword (low-order word first) that contains the number of elapsed ticks.

7.3.22   SYS_TICKFREQ (0x31)

Returns the tick frequency.

Entry

Register r1 must contain 0 on entry to this routine.

Return

On exit, r0 contains either:

*   the number ticks per second

*   –1 if the target does not know the value of one tick. RVI-ME does not support this SWI and always returns –1 in r0.

7.4 Debug agent interaction SWIs

In addition to the C library semihosted functions described in *Semihosting SWIs* on page 7-6, the following SWIs support interaction with the debug agent:

EnterSVC   This SWI sets the processor to Supervisor mode. See *angel_SWIreason_EnterSVC (0x17)*.

ReportException

This SWI is used to report an exception to the debugger. See *angel_SWIreason_ReportException (0x18)* on page 7-21.

——— **Note** ———

The reason_LateStartup SWI was supported in previous releases of RVCT. This is now obsolete and no longer supported.

7.4.1 angel_SWIreason_EnterSVC (0x17)

Sets the processor to Supervisor (SVC) mode and disables all interrupts by setting both interrupt mask bits in the new CPSR. With RVI-ME, the User stack pointer (r13_USR) is copied to the Supervisor stack pointer (r13_SVC) and the I and F bits in the current CPSR are set, disabling normal and fast interrupts.

Entry

Register r1 is not used. The CPSR can specify User or Supervisor mode.

Return

On exit, r0 contains the address of a function to be called to return to User mode. The function has the following prototype:

void ReturnToUSR(**void**)

If EnterSVC is called in User mode, this routine returns the caller to User mode and restores the interrupt flags. Otherwise, the action of this routine is undefined.

If entered in User mode, the Supervisor stack is lost as a result of copying the user stack pointer. The return to User routine restores r13_SVC to the Angel Supervisor mode stack value, but this stack must not be used by applications.

After executing the SWI, the current link register is r14_SVC, not r14_USR. If the value of r14_USR is required after the call, it must be pushed onto the stack before the call and popped afterwards, as for a BL function call.

7.4.2 angel_SWIreason_ReportException (0x18)

This SWI can be called by an application to report an exception to the debugger directly. The most common use is to report that execution has completed, using `ADP_Stopped_ApplicationExit`.

Entry

On entry `r1` is set to one of the values listed in Table 7-3 and Table 7-4 on page 7-22. These values are defined in `adp.h`.

The hardware exceptions are generated if the debugger variable `vector_catch` is set to catch that exception type, and the debug agent is capable of reporting that exception type.

Table 7-3 Hardware vector reason codes

Name (#defined in adp.h)	Hexadecimal value
`ADP_Stopped_BranchThroughZero`	0x20000
`ADP_Stopped_UndefinedInstr`	0x20001
`ADP_Stopped_SoftwareInterrupt`	0x20002
`ADP_Stopped_PrefetchAbort`	0x20003
`ADP_Stopped_DataAbort`	0x20004
`ADP_Stopped_AddressException`	0x20005
`ADP_Stopped_IRQ`	0x20006
`ADP_Stopped_FIQ`	0x20007

Exception handlers can use these SWIs at the end of handler chains as the default action, to indicate that the exception has not been handled.

Table 7-4 Software reason codes

Name (#defined in adp.h)	Hexadecimal value
ADP_Stopped_BreakPoint	0x20020
ADP_Stopped_WatchPoint	0x20021
ADP_Stopped_StepComplete	0x20022
ADP_Stopped_RunTimeErrorUnknown	*0x20023
ADP_Stopped_InternalError	*0x20024
ADP_Stopped_UserInterruption	0x20025
ADP_Stopped_ApplicationExit	0x20026
ADP_Stopped_StackOverflow	*0x20027
ADP_Stopped_DivisionByZero	*0x20028
ADP_Stopped_OSSpecific	*0x20029

In Table 7-4, a * next to a value indicates that the value is not supported by ARM debuggers. The debugger reports an `Unhandled ADP_Stopped exception` for these values.

Return

No return is expected from these calls. However, it is possible for the debugger to request that the application continue. In this case, execution continues with the registers as they were on entry to the SWI, or as subsequently modified by the debugger.

Appendix A
Via File Syntax

This appendix describes the syntax of via files accepted by the ARM® development tools, such as the compiler, linker, assembler, and librarian utility. It contains the following sections:

- *Overview of via files* on page A-2
- *Syntax* on page A-3.

A.1 Overview of via files

Via files are plain text files that contain command-line arguments and options to ARM development tools. You can use via files with most of the ARM command-line tools, including the:

- compiler, armcc
- assembler, armasm
- linker, armlink
- librarian, armar.

You can specify a via file from the command line using the --via command-line option. See the documentation for the individual tool for more information.

In general, you can use a via file to specify any command-line option to a tool, including --via. This means that you can call multiple nested via files from within a via file.

A.1.1 Via file evaluation

When a tool that supports via files is invoked it:

1. Scans for arguments that cause all other arguments to be ignored, such as --help and --vsn.

   If such an argument is found, via files are not processed.

2. Replaces the first specified --via *via_file* argument with the sequence of argument words extracted from the via file, including recursively processing any nested --via commands in the via file.

3. Processes any subsequent --via *via_file* arguments in the same way, in the order they are presented.

That is, via files are processed in the order you specify them, and each via file is processed completely, including processing nested via files, before processing the next via file.

A.2   Syntax

Via files must conform to the following syntax rules:

* A via file is a text file containing a sequence of words. Each word in the text file is converted into an argument string and passed to the tool.

* Words are separated by whitespace, or the end of a line, except in delimited strings. For example:

  `--c90 --strict` (two words)`--c90--strict` (one word)

* The end of a line is treated as whitespace. For example:

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

  is equivalent to:

  `--c90 --strict`

* Strings enclosed in quotation marks ("), or apostrophes (') are treated as a single word. Within a quoted word, an apostrophe is treated as an ordinary character. Within an apostrophe delimited word, quote is treated as an ordinary character.

  Quotation marks are used to delimit filenames or path names that contain spaces. For example:

  `-I C:\My Project\includes` (three words)`-I "C:\My Project\includes"` (two words)

  Apostrophes can be used to delimit words that contain quotes. For example:

  `-DNAME='"RealView Compilation Tools"'` (one word)

* Characters enclosed in parentheses are treated as a single word. For example:

  `--option(x, y, z)` (one word)`--option (x, y, z)` (two words)

* Within quoted or apostrophe delimited strings, you can use a backslash (\) character to escape the quote, apostrophe, and backslash characters.

* A word that occurs immediately next to a delimited word is treated as a single word. For example:

  `-I"C:\Project\includes"`

  is treated as the single word:

  `-IC:\Project\includes`

- Lines beginning with a semicolon (;) or a hash (#) character as the first non-whitespace character are comment lines. If a semicolon or hash character appears anywhere else in a line, it is not treated as the start of a comment. For example:

  ```
  -o objectname.axf    ;this is not a comment
  ```

  A comment ends at the end of a line, or at the end of the file. There are no multi-line comments, and there are no part-line comments.

- Lines that include the preprocessor option -Dsymbol="value" must be delimited with a single quote, either as '-D*symbol*="*value*"' or as -Dsymbol='"value"'. For example:

  ```
  -c -DFOO_VALUE='"FOO_VALUE"'
  ```

Appendix B
Standard C Implementation Definition

This appendix gives information required by the ISO C standard for conforming C implementations. It contains the following section:

- *Implementation definition* on page B-2.

B.1 Implementation definition

Appendix G of the ISO C standard (ISO/IEC 9899:1990 (E)) collates information about portability issues. Subclause G3 lists the behavior that each implementation must document.

— **Note** —

This appendix does not duplicate information that is part of the compiler-specific implementations (see *Compiler-specific features* on page 3-2). This appendix provides references where applicable.

The following subsections correspond to the relevant sections of subclause G3. They describe aspects of the ARM® C compiler and C library, not defined by the ISO C standard, that are implementation-defined:

- *Translation* on page B-3
- *Environment* on page B-3
- *Identifiers* on page B-5
- *Characters* on page B-5
- *Integers* on page B-5
- *Floating-point* on page B-5
- *Arrays and pointers* on page B-5
- *Registers* on page B-6
- *Structures, unions, enumerations, and bitfields* on page B-6
- *Qualifiers* on page B-7
- *Declarators* on page B-7
- *Statements* on page B-7
- *Preprocessing directives* on page B-8
- *Library functions* on page B-8.

— **Note** —

The support for the wctype.h and wchar.h headers excludes wide file operations.

B.1.1 Translation

Diagnostic messages (see *Diagnostic messages* on page 2-58) produced by the compiler are of the form:

source-file, *line-number*: *severity*: *error-code*: *explanation*

where *severity* is one of:

[blank]   If the severity is blank, this is a remark and indicates common, but sometimes unconventional, use of C or C++. Remarks are not displayed by default. Use the --remarks option to display remark messages (see *Controlling the output of diagnostic messages* on page 2-60). Compilation continues.

Warning   Indicates unusual conditions in your code that might indicate a problem. Compilation continues.

Error   Indicates a problem that causes the compilation to stop. For example, violations in the syntactic or semantic rules of the C or C++ language.

Internal Error

Indicates an internal problem with the compiler. Contact your supplier with the details listed in *Feedback on RealView Developer Kit* on page xiii.

Here:

error-code   Is a number identifying the error type.

explanation Is a text description of the error.

B.1.2 Environment

The mapping of a command line from the ARM architecture-based environment into arguments to main() is implementation-specific. The generic ARM C library supports the following:

- *main()*
- *Interactive device* on page B-4
- *Redirecting standard input, output, and error streams* on page B-4.

main()

The arguments given to main() are the words of the command line (not including input/output redirections), delimited by whitespace, except where the whitespace is contained in double quotes.

Note

- A whitespace character is any character where the result of isspace() is true.

- A double quote or backslash character \ inside double quotes must be preceded by a backslash character.

- An input/output redirection is not recognized inside double quotes.

Interactive device

In an nonhosted implementation of the ARM C library, the term *interactive device* might be meaningless. The generic ARM C library supports a pair of devices, both called :tt, intended to handle keyboard input and VDU screen output. In the generic implementation:

- no buffering is done on any stream connected to :tt unless input/output redirection has occurred

- if input/output redirection other than to :tt has occurred, full file buffering is used (except that line buffering is used if both stdout and stderr were redirected to the same file).

Redirecting standard input, output, and error streams

Using the generic ARM C library, the standard input (stdin), output (stdout) and error streams (stderr) can be redirected at runtime. For example, if mycopy is a program, running on a host debugger, that copies the standard input to the standard output, the following line runs the program:

```
mycopy < infile > outfile 2> errfile
```

and redirects the files as follows:

stdin   The file is redirected to infile.

stdout   The file is redirected to outfile.

stderr   The file is redirected to errfile.

The permitted redirections are:

0< *filename*   This reads stdin from *filename*.

< *filename*   This reads stdin from *filename*.

1> *filename*   This writes stdout to *filename*.

> *filename*   This writes stdout from *filename*.

2> *filename*   This writes stderr from *filename*.

2>&1   This writes stderr to the same place as stdout.

>& *file*   This writes both stdout and stderr to *filename*.

>> *filename*   This appends stdout to *filename*.

>>& *filename* This appends both stdout and stderr to *filename*.

To redirect stdin, stdout, and stderr on the target, you must define:

#pragma import(_main_redirection)

File redirection is done only if either:

- the invoking operating system supports it

- the program reads and writes characters and has not replaced the C library functions fputc() and fgetc().

B.1.3   Identifiers

See *Character sets and identifiers* on page 3-47 for details.

B.1.4   Characters

See *Character sets and identifiers* on page 3-47 for details.

B.1.5   Integers

See *Integer* on page 3-50 for details.

B.1.6   Floating-point

See *Float* on page 3-50 for details.

B.1.7   Arrays and pointers

See *Arrays and pointers* on page 3-51 for details.

B.1.8   Registers

Using the ARM compiler, you can declare any number of local objects (auto variables) to have the storage class **register**. See *Variable declaration keywords* on page 3-21 for information on how the ARM compiler implements the **register** storage class.

B.1.9   Structures, unions, enumerations, and bitfields

The ISO/IEC C standard requires the following implementation details to be documented for structured data types:

*   the outcome when a member of a union is accessed using a member of different type

*   the padding and alignment of members of structures

*   whether a plain **int** bitfield is treated as a **signed int** bitfield or as an **unsigned int** bitfield

*   the order of allocation of bitfields within a unit

*   whether a bitfield can straddle a storage-unit boundary

*   the integer type chosen to represent the values of an enumeration type.

These implementation details are documented in the relevant sections of *C and C++ implementation details* on page 3-47.

Unions

See *Unions* on page 3-53 for details.

Enumerations

See *Enumerations* on page 3-53 for details.

Padding and alignment of structures

See *Structures* on page 3-54 for details.

Bitfields

See *Bitfields* on page 3-55 for details.

B.1.10 Qualifiers

An object that has a volatile-qualified type is accessed if any word or byte (or halfword on ARM architectures that have halfword support) of it is read or written. For volatile-qualified objects, reads and writes occur as directly implied by the source code, in the order implied by the source code.

The effect of accessing a volatile-qualified `short` is undefined on ARM architectures that do not have halfword support.

B.1.11 Declarators

The number of declarators that can modify an arithmetic, structure, or union type is limited only by available memory.

B.1.12 Statements

The number of case values in a `switch` statement is limited only by memory.

Expression evaluation

The compiler performs the usual arithmetic conversions (promotions) set out in the appropriate C or C++ standard before evaluating an expression.

———— **Note** ————

- The compiler can re-order expressions involving only associative and commutative operators of equal precedence, even in the presence of parentheses. For example, a + (b – c) might be evaluated as (a + b) – c if a, b, and c are integer expressions.

- Between sequence points, the compiler can evaluate expressions in any order, regardless of parentheses. Therefore, side effects of expressions between sequence points can occur in any order.

- The compiler can evaluate function arguments in any order.

Any aspect of evaluation order not prescribed by the relevant standard, can vary between releases of the ARM compiler.

B.1.13 Preprocessing directives

The ISO standard C header files can be referred to as described in the standard, for example, #include <stdio.h>.

Quoted names for includable source files are supported. The compiler accepts host filenames.

The recognized #pragma directives are shown in *Pragmas* on page 3-2.

B.1.14 Library functions

The ISO C library variants are listed in *About the runtime libraries* on page 5-2.

The precise nature of each C library is unique to the particular implementation. The generic ARM C library has, or supports, the following features:

• The macro NULL expands to the integer constant 0.

• If a program redefines a reserved external identifier such as printf, an error might occur when the program is linked with the standard libraries. If it is not linked with standard libraries, no error is detected.

• The assert() function prints the following message on stderr and then calls the abort() function:

  *** assertion failed: *expression*, file *name*, line *number*

For implementation details of mathematical functions, locale, signals, and input/output see Chapter 5 *The C and C++ Libraries*.

Appendix C
Standard C++ Implementation Definition

The ARM® compiler supports the majority of the language features described in the ISO/IEC standard for C++ when compiling C++. This appendix lists the C++ language features defined in the standard, and states whether or not that language feature is supported by ARM C++. It contains the following sections:

- *Integral conversion* on page C-2
- *Calling a pure virtual function* on page C-3
- *Major features of language support* on page C-4
- *Standard C++ library implementation definition* on page C-5.

——— **Note** ———

This appendix does not duplicate information that is part of the standard C implementation. See Appendix B *Standard C Implementation Definition*.

When compiling C++ in ISO C mode, the ARM compiler is identical to the ARM C compiler. Where there is an implementation feature specific to either C or C++, this is noted in the text. For extension to standard C++, see *Language extensions* on page 3-30.

C.1 Integral conversion

During integral conversion, if the destination type is signed, the value is unchanged if it can be represented in the destination type and bitfield width. Otherwise, the value is truncated to fit the size of the destination type.

—— **Note** ——

This section is related to section 4.7 of the ISO/IEC standard.

C.2   Calling a pure virtual function

Calling a pure virtual function is illegal. If your code calls a pure virtual function, then the compiler includes a call to the library function `__cxa_pure_virtual`. You must not call this function directly.

`__cxa_pure_virtual` raises the signal **SIGPVFN**. The default signal handler prints an error message and exits. See *__default_signal_handler()* on page 5-75.

Compile the code in Example C-1. Run the program in your debugger to see the effect of this call.

Also, see *Defining optimization criteria* on page 2-43 for a description of the optimization options available for unused virtual function elimination.

Example C-1 Calling a pure virtual function

```
struct B {
    B();
    virtual void f() = 0; // B::f is pure virtual
};
void g(B* b) { b->f(); }
B::B() {
    g(this); // calls 'f' while B is being constructed
}
struct D : B {
    virtual void f() { }
};
int main() {
    D d;
    return 0;
}
```

C.3   Major features of language support

Table C-1 shows the major features of the language supported by this release of ARM C++.

Table C-1 Major feature support for language

Major feature	ISO/IEC standard section	Support
Core language	1 to 13	Yes.
Templates	14	Yes, with the exception of export templates.
Exceptions	15	Yes.
Libraries	17 to 27	See the *Standard C++ library implementation definition* on page C-5 and Chapter 5 *The C and C++ Libraries*.

C.4   Standard C++ library implementation definition

Version 2.02.03 of the Rogue Wave library provides a subset of the library defined in the standard. There are small differences from the December 1996 version of the ISO/IEC standard. For details of the implementation definition, see *Standard C++ library implementation definition* on page 5-118.

The library can be used with user-defined functions to produce target-dependent applications. See *About the runtime libraries* on page 5-2 for more information.

Appendix D
C and C++ Compiler Implementation Limits

This appendix lists the implementation limits when using the ARM® compiler to compile C and C++. It contains the following sections:

- *C++ ISO/IEC standard limits* on page D-2
- *Internal limits* on page D-4
- *Limits for integral numbers* on page D-5
- *Limits for floating-point numbers* on page D-6.

D.1 C++ ISO/IEC standard limits

The ISO/IEC C++ standard recommends minimum limits that a conforming compiler must accept. You must be aware of these when porting applications between compilers. Table D-1 gives a summary of these limits.

In this table, a limit of `memory` indicates that the ARM compiler imposes no limit, other than that imposed by the available memory.

Table D-1 Implementation limits

Description	Recommended	ARM
Nesting levels of compound statements, iteration control structures, and selection control structures.	256	memory
Nesting levels of conditional inclusion.	256	memory
Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration.	256	memory
Nesting levels of parenthesized expressions within a full expression.	256	memory
Number of initial characters in an internal identifier or macro name.	1024	memory
Number of initial characters in an external identifier.	1024	memory
External identifiers in one translation unit.	65536	memory
Identifiers with block scope declared in one block.	1024	memory
Macro identifiers simultaneously defined in one translation unit.	65536	memory
Parameters in one function declaration. Overload resolution is sensitive to the first 32 arguments only.	256	memory
Arguments in one function call. Overload resolution is sensitive to the first 32 arguments only.	256	memory
Parameters in one macro definition.	256	memory
Arguments in one macro invocation.	256	memory
Characters in one logical source line.	65536	memory
Characters in a character string literal or wide string literal after concatenation.	65536	memory
Size of a C or C++ object (including arrays).	262144	4294967296

Table D-1 Implementation limits

Description	Recommended	ARM
Nesting levels of #include file.	256	memory
Case labels for a switch statement, excluding those for any nested switch statements.	16384	memory
Data members in a single class, structure, or union.	16384	memory
Enumeration constants in a single enumeration.	4096	memory
Levels of nested class, structure, or union definitions in a single struct-declaration-list.	256	memory
Functions registered by `atexit()`.	32	33
Direct and indirect base classes.	16384	memory
Direct base classes for a single class.	1024	memory
Members declared in a single class.	4096	memory
Final overriding virtual functions in a class, accessible or not.	16384	memory
Direct and indirect virtual bases of a class.	1024	memory
Static members of a class.	1024	memory
Friend declarations in a class.	4096	memory
Access control declarations in a class.	4096	memory
Member initializers in a constructor definition.	6144	memory
Scope qualifications of one identifier.	256	memory
Nested external specifications.	1024	memory
Template arguments in a template declaration.	1024	memory
Recursively nested template instantiations.	17	memory
Handlers per try block.	256	memory
Throw specifications on a single function declaration.	256	memory

D.2 Internal limits

In addition to the limits described in Table D-1 on page D-2, the compiler has internal limits as listed in Table D-2.

Table D-2 Internal limits

Description	ARM
Maximum number of lines in a C source file. (The internal representation of a translation unit, or all translation units if using `--multifile`, must fit in store.)	memory
Maximum number of virtual registers.	65536
Number of characters in a mangled name before it is truncated.	4096
Number of bits in the smallest object that is not a bit field (`CHAR_BIT`).	8
Maximum number of bytes in a multibyte character, for any supported locale (`MB_LEN_MAX`).	1

D.3 Limits for integral numbers

Table D-3 gives the ranges for integral numbers in ARM C and C++. The Endpoint column of the table gives the numerical value of the range endpoint. The Hex value column gives the bit pattern (in hexadecimal) that is interpreted as this value by the ARM compiler. These constants are defined for you in limits.h include file.

When entering a constant, choose the size and sign with care. Constants are interpreted differently in decimal and hexadecimal/octal. See the appropriate C or C++ standard, or any of the recommended C and C++ textbooks for more details (see *Further reading* on page x).

Table D-3 Integer ranges

Constant	Meaning	Endpoint	Hex value
CHAR_MAX	Maximum value of **char**	255	0xFF
CHAR_MIN	Minimum value of **char**	0	0x00
SCHAR_MAX	Maximum value of **signed char**	127	0x7F
SCHAR_MIN	Minimum value of **signed char**	−128	0x80
UCHAR_MAX	Maximum value of **unsigned char**	255	0xFF
SHRT_MAX	Maximum value of **short**	32767	0x7FFF
SHRT_MIN	Minimum value of **short**	−32768	0x8000
USHRT_MAX	Maximum value of **unsigned short**	65535	0xFFFF
INT_MAX	Maximum value of **int**	2147483647	0x7FFFFFFF
INT_MIN	Minimum value of **int**	−2147483648	0x80000000
LONG_MAX	Maximum value of **long**	2147483647	0x7FFFFFFF
LONG_MIN	Minimum value of **long**	−2147483648	0x80000000
ULONG_MAX	Maximum value of **unsigned long**	4294967295	0xFFFFFFFF
LLONG_MAX	Maximum value of **long long**	9.2E+18	0x7FFFFFFF FFFFFFFF
LLONG_MIN	Minimum value of **long long**	−9.2E+18	0x80000000 00000000
ULLONG_MAX	Maximum value of **unsigned long long**	1.8E+19	0xFFFFFFFF FFFFFFFF

D.4 Limits for floating-point numbers

Table D-4 and Table D-5 on page D-7 give the characteristics, ranges, and limits for floating-point numbers. These constants are defined for you in `limits.h` include file.

Table D-4 Floating-point limits

Constant	Meaning	Value
FLT_MAX	Maximum value of **float**	3.40282347e+38F
FLT_MIN	Minimum normalized positive floating-point number of **float**	1.17549435e–38F
DBL_MAX	Maximum value of **double**	1.79769313486231571e+308
DBL_MIN	Minimum normalized positive floating-point number value of **double**	2.22507385850720138e–308
LDBL_MAX	Maximum value of **long double**	1.79769313486231571e+308
LDBL_MIN	Minimum normalized positive floating-point number value of **long double**	2.22507385850720138e–308
FLT_MAX_EXP	Maximum value of base 2 exponent for type **float**	128
FLT_MIN_EXP	Minimum value of base 2 exponent for type **float**	−125
DBL_MAX_EXP	Maximum value of base 2 exponent for type **double**	1024
DBL_MIN_EXP	Minimum value of base 2 exponent for type **double**	−1021
LDBL_MAX_EXP	Maximum value of base 2 exponent for type **long double**	1024
LDBL_MIN_EXP	Minimum value of base 2 exponent for type **long double**	−1021
FLT_MAX_10_EXP	Maximum value of base 10 exponent for type **float**	38
FLT_MIN_10_EXP	Minimum value of base 10 exponent for type **float**	−37
DBL_MAX_10_EXP	Maximum value of base 10 exponent for type **double**	308
DBL_MIN_10_EXP	Minimum value of base 10 exponent for type **double**	−307
LDBL_MAX_10_EXP	Maximum value of base 10 exponent for type **long double**	308
LDBL_MIN_10_EXP	Minimum value of base 10 exponent for type **long double**	−307

Table D-5 Other floating-point characteristics

Constant	Meaning	Value
FLT_RADIX	Base (radix) of the ARM floating-point number representation	2
FLT_ROUNDS	Rounding mode for floating-point numbers	(nearest) 1
FLT_DIG	Decimal digits of precision for **float**	6
DBL_DIG	Decimal digits of precision for **double**	15
LDBL_DIG	Decimal digits of precision for **long double**	15
FLT_MANT_DIG	Binary digits of precision for type **float**	24
DBL_MANT_DIG	Binary digits of precision for type **double**	53
LDBL_MANT_DIG	Binary digits of precision for type **long double**	53
FLT_EPSILON	Smallest positive value of x that 1.0 + x != 1.0 for type **float**	1.19209290e–7F
DBL_EPSILON	Smallest positive value of x that 1.0 + x != 1.0 for type **double**	2.2204460492503131e–16
LDBL_EPSILON	Smallest positive value of x that 1.0 + x != 1.0 for type **long double**	2.2204460492503131e–16L

———— **Note** ————

- When a floating-point number is converted to a shorter floating-point number, it is rounded to the nearest representable number.

- Floating-point arithmetic conforms to IEEE 754.

Appendix E
Using Older Options

This appendix describes command-line options that have changed between previous releases of *RealView® Compilation Tools*, that is RVCT v2.1, RVCT v2.0, and RVCT v1.2.

Many of the options listed here were supported in ADS v1.2 but are no longer supported in the RVCT tools suite.

This appendix contains the following section:

- *Mapping old compiler options to the new release* on page E-2.

E.1   Mapping old compiler options to the new release

Table E-1 describes options used by ARM compilers in previous releases of RVCT. If you are upgrading from a previous release of RVCT, use Table E-1 to choose replacement options. If you are upgrading from ADS v1.2, use Table E-1 to identify those options that are no longer supported in the RVCT tools suite, and to find new names for these flags in the latest release.

———— **Note** ————

If you are upgrading to RVCT from a previous release, you must migrate your use of these options to the equivalent options in the latest release of the compiler.

All the options listed in Table E-1 are obsolete in the latest release of RVCT and no longer supported. However, in some cases, a warning message is displayed to describe the preferred option (see *Warning messages in RVCT v2.2* on page E-4).

Table E-1 Mapping of compiler options

Old compiler option	Equivalent compiler option	Usage notes
-ansi	--c90	
-ansic	--c90	
-auto_float_constants	--fpmode std	
-dll_vtbl	--export_vtbl_all	
-Ec	--loose_implicit_cast	
-fa	None	Data flow analysis feature on by default since RVCT v2.1
-fd	--sys_include	
-fi	None	
-fj	None	
-fk	--kandr_include	
-fp	None	
-fu	None	
-fv	None	

Table E-1 Mapping of compiler options

Old compiler option	Equivalent compiler option	Usage notes
-fy	--enum_is_int	
-fx	--remarks or --strict_warnings	
-g-	--no_debug	
-g+	-g or --debug	
-gtp (or its synonym -gt-p)	--no_debug_macros	
-gt+p	--debug_macros	
-list	--list	Option name is the same but the behavior has changed
--no_inlinemax	--forceinline	
-Oautoinline	--autoinline	
-Odata_reorder	--data_reorder	
-Oinline	--inline	
-Oldrd, --Ono_ldrd	None	Selected automatically since RVCT v2.1
-Ono_autoinline	--no_autoinline	
-Ono_data_reorder	--no_data_reorder	
-Ono_inline	--no_inline	
-W*letter*	--diag_suppress *tag*	Use *tag* to identify the number of the warning message to suppress
-E*letter*	None	
-zas*number*	__align	
-zat	--min_array_alignment or __align	
-zc	--signed_chars	
-zo	--split_sections	

E.1.1   Warning messages in RVCT v2.2

If you are using the latest release of RVCT and you specify a command-line option where a preferred option name exists, the compiler might issue a warning. For example:

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

                                           ARM DUI 0282B

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 architecture-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.

American National Standards Institute (ANSI)

An organization that specifies standards for, among other things, computer software. This is superseded by the International Standards Organization.

Anachronisms   Various C++ language features that are no longer strictly legal.

ANSI   *See* American National Standards Institute.

API   *See* Application Programming Interface.

Application Programming Interface (API)

The syntax of the functions and procedures within a module or library.

Architecture   The term used to identify a group of processors that have similar characteristics.

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 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.

ARM Toolkit Proprietary ELF (ATPE)

The binary file format used by RealView Developer Kit. ATPE object format is produced by the compiler and assembler tools. The ARM linker accepts ATPE object files and can output an ATPE executable file. RealView Debugger can load only ATPE format images, or binary ROM images produced by the fromELF utility.

See also RealView Developer Suite.

ATPE   *See* ARM Toolkit Proprietary ELF.

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

Byte   A unit of memory storage consisting of eight bits.

Char   A unit of storage for a single character. ARM designs use a byte to store a single character and an integer to store two to four characters.

Class   A C++ class involved in the image.

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

Current place   In compiler terminology, the directory that contains files to be included in the compilation process.

Debug With Arbitrary Record Format (DWARF)

ARM code generation tools generate debug information in DWARF2 format by default. From RVCT v2.2, you can optionally generate DWARF3 format (Draft Standard 9).

Debugger	An application that monitors and controls the execution of a second application. Usually used to find errors in the application program flow.
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.
Doubleword	A 64-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
DWARF	*See* Debug With Arbitrary Record Format.
ELF	*See* Executable and Linking Format.

Embedded assembler

Embedded assembler is assembler code that is included in a function, and is separate from other C or C++ functions.

See also Inline.

Environment	The actual hardware and operating system that an application runs on.

Executable and Linking Format (ELF)

The industry standard binary file format used by RealView Developer Kit. 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 a partially linked ELF object.

Execution view	The address of regions and sections after the image has been loaded into memory and started execution.
Flash memory	Nonvolatile memory that is often used to hold application code.
Globals	Variables or functions with the image with global scope.
Halfword	A 16-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
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.
IDE	Integrated Development Environment (for example, the ARM RealView Debugger IDE).
Image	An executable file that has been loaded onto a processor for execution.
	A binary execution file loaded onto a processor and given a thread of execution. An image can have multiple threads. An image is related to the processor that is running the default thread for the image.

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 assembler.

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

Interrupt   A change in the normal processing sequence of an application caused by, for example, an external signal.

Interworking   Producing an application that uses both ARM and Thumb code.

ISO   *See* International Standards Organization.

Jazelle   The Jazelle architecture extends the existing ARM architecture to enable direct execution of selected JVM (Java Virtual Machine) opcodes.

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 where the least significant byte of a word is at a lower address than the most significant byte.

Load view   The address of regions and sections when the image has been loaded into memory but has not yet started execution.

PCH   Precompiled Header

PIC   Position Independent Code.

See also ROPI.

PID   Position Independent Data.

See also RWPI.

Precompiled Header (PCH)
                               A header file that is precompiled. This avoids the compiler having to compile the file each time it is included by source files.

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.

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 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 Kit (RVDK)
RealView Developer Kit is a suite of software development applications, together with supporting documentation and examples, that enables you to write and debug 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.

RealView ICE Micro Edition (RVI-ME)
A JTAG-based debug tool for embedded systems.

Redirection The process of sending default output to a different destination or receiving default input from a different source. This is commonly used to output text, that would otherwise be displayed on the computer screen, to a file.

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.

Reentrancy The ability of a subroutine to have more that one instance of the code active. Each instance of the subroutine call has its own copy of any required static data.

Remapping Changing the address of physical memory or devices after the application has started executing. This is typically done to enable RAM to replace ROM after the initialization has been done.

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

RISC *See* Reduced Instruction Set Computing.

ROPI *See* Read-Only Position Independent.

RTOS	Real Time Operating System.
RVCT	*See* RealView Compilation Tools.
RVDK	*See* RealView Developer Kit.
RVDS	*See* RealView Developer Suite.
RVI	*See* RealView ICE.
RVI-ME	*See* RealView ICE Micro Edition.
RWPI	*See* Read Write Position Independent.
Scatter-loading	Assigning the address and grouping of code and data sections individually rather than using single large blocks.
Scope	The accessibility of a function or variable at a particular point in the application code. Symbols that have global scope are always accessible. Symbols with local or private scope are only accessible to code in the same subroutine or object.
Semihosting	A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather attempting to support the I/O itself.
Signal	An indication of abnormal processor operation.
Software Interrupt (SWI)	
	An instruction that causes the processor to call a programmer-specified subroutine. Used by ARM to handle semihosting.
Stack	The portion of memory that is used to record the return address of code that calls a subroutine. The stack can also be used for parameters and temporary variables.
SWI	*See* Software Interrupt.
Target	The actual target processor, (real or simulated), that is running the target application.
	The fundamental object in any debugging session. The basis of the debugging system. The environment in which the target software runs. It is essentially a collection of real or simulated processors.
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.
Vector Floating Point (VFP)	
	A standard for floating-point coprocessors where several data values can be processed by a single instruction.

Veneer

A small block of code used with subroutine calls when there is a requirement to change processor state or branch to an address that cannot be reached in the current processor state.

VFP

See Vector Floating Point.

Volatile

Memory addresses where the contents can change independently of the executing application are described as volatile. These are typically memory-mapped peripherals.

See also Memory mapped

Word

A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

Zero Initialized (ZI)

R/W memory used to hold variables that do not have an initial value. The memory is normally set to zero on reset.

ZI

See Zero Initialized.

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.

D

Data flow analysis
  about 2-63
  checking for anomalies 2-63
Data sections
  controlling 2-51
__DATE__, C and C++ macro 3-90
Debug agent
  interaction, semihosting SWIs 7-20
Debug agent interaction SWIs 7-20
Debug interaction SWIs 7-20
  angel_SWIreason_EnterSVC 7-20
  angel_SWIreason_ReportException
    7-21
  overview 7-20
Debug tables
  compiler options 2-41
  DWARF2 format 2-42
  DWARF3 format 2-42
  increase size 2-42
  reduce size 2-42
Default object extension 2-57
Default version
  symbol versioning 2-56
__default_signal_handler() 5-73
Denormal 6-38
Diagnostic messages
  arm style 2-60
  changing severity of 2-61
  compiler options 2-27, 2-58, 2-60,
    2-61
  controlled by pragmas 3-9
  controlling 2-58
  controlling output of 2-60
  controlling with pragmas 3-8
  ide style 2-60
  severity 2-59
  suppressing 2-62
Digraph tokens
  recognizing 2-54
Directives
  EQU 7-4
  PRESERVE8 3-28
  REQUIRE8 3-28
__disable_fiq() intrinsic, C and C++
  3-18
__disable_irq() intrinsic, C and C++
  3-18

DWARF2
  debug table format 2-42
  support 1-3
DWARF3
  debug table format 2-42
  support 1-3

E

__EDG__, C and C++ macro 3-90
__EDG_IMPLICIT_USING_STD, C++ macro
  3-90
__EDG_VERSION__, C and C++ macro
  3-90
Eight-byte alignment
  __align 3-23
  alloca() 5-124
  compatibility with legacy code 2-64
  heap implementations 5-78
  __Heap_Alloc() 5-83
  __Heap_ProvideMemory() 5-82
  __Heap_Realloc() 5-84
  PRESERVE8 3-28
  REQUIRE8 3-28
  __rt_heap_extend() 5-93
  summary of features 3-28
  __user_heap_extend() 5-90
  __user_initial_stackheap() 5-89
Embedded assembler 4-20–4-28
  ARMv6 instructions 4-28
  asm keyword 4-20
  __asm keyword 4-20
  calling non-static member functions
    4-27
  calling non-virtual member
    functions 4-27
  calling virtual member functions
    4-28
  __cpp 4-25
  differences from older compilers
    4-28
  differences with C/C++ 4-23
  differences with inline assembler
    4-31
  generating embedded functions
    4-23
  manual overload resolution 4-25
  member function classes 4-26

  __offsetof_base 4-25
  related base classes 4-25
  restrictions 4-22
  syntax 4-20
Embedded assemblers 2-8, 4-20
Embedded assembly
  about 2-8
__enable_fiq() intrinsic, C and C++
  3-18
__enable_irq() intrinsic, C and C++
  3-18
Endianness
  in the compiler 2-43
Enumerations
  controlling 2-54
enum, C and C++ keyword 3-53
Environment variables
  ARMINC 2-11, 2-12
  ARMLIB 5-4
  RVCT22INC 2-11, 2-12
  RVCT22LIB 5-4
  RVCT22_CCOPT 2-22
  TMP 2-13
  TMPDIR 2-13
Epsilon 6-38
EQU
  directive 7-4
Error
  meaning 2-59
Error messages
  implicit cast 2-55
  library 5-71
  redirecting 2-26
  specifying additional checks 2-55
  tailoring handling 5-71
Examples
  main directory 1-2
Exception handling, C++ 3-63
Exception unwinding
  see Function unwinding
Exceptions
  and debug agent 7-21
__EXCEPTIONS, C and C++ macro 3-90
Exceptions, C++ C-4
  see Function unwinding
Execution
  tailoring the C library to a new
    environment 5-35
Exit status

ARM DUI 0282B