RealView® Developer Kit

Version 1.0

Compiler and Libraries Guide

ARM®

RealView Developer Kit
Compiler and Libraries Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
May 2003	A	Release for RVDK v1.0
March 2004	B	Release for RVDK v1.0.1

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 v1.0 Compiler and Libraries Guide*. It contains the following sections:

- *About this book* on page viii
- *Feedback* on page xiii.

About this book

This book provides reference information for *RealView Compilation Tools* (RVCT) v2.0, 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 libraries.

Intended audience

This book is written for all developers who are producing applications using RVCT v2.0. It assumes that you are an experienced software developer. For an overview of the ARM development tools provided with RVDK v1.0, see your Getting Started guide.

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 RVCT v2.0 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 RVCT v2.0.

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/IEC C standards 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 *Older Compiler Options*

Read this appendix for information on the older compiler options that are supported in the ARM compiler. It also shows you the equivalent options to use in the new ARM compiler, and the equivalents to the older messages that are specified in this appendix.

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, 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 v1.0 Compiler and Libraries Guide* (ARM DUI 0232)
- *RealView Developer Kit v1.0 Assembler Guide* (ARM DUI 0233)
- *RealView Developer Kit v1.0 Debugger User Guide* (ARM DUI 0234)
- *RealView Developer Kit v1.0 Debugger Command Line Reference Guide* (ARM DUI 0235).
- *RealView ICE Micro Edition v1.1 User Guide* (ARM DUI 0220).

The following additional documentation is provided with RealView Developer Kit:

- *ARM FLEXlm License Management Guide* (ARM DUI 0209).

Refer to the ARM web site for additional information and documentation relating to ARM products.

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.

The following book gives general information about the ARM architecture:

- *ARM system-on-chip architecture* (second edition), Furber, S., (2000). Addison Wesley. ISBN 0-201-67519-6.

The following books describe the C++ language:

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

- Stroustrup, B., *The C++ Programming Language* (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.

- 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++* (1996). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-63371-X.

  The sequel to *Effective C++*.

The following books provide general C programming information:

- 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* (second edition, 1987). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-109802-0.

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

- 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.

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

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

Feedback

ARM Limited welcomes feedback on both RealView Developer Kit and the documentation.

Feedback on RealView Developer Kit

If you have any problems with RealView Developer Kit, 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 v1.0 (RVDK v1.0). It contains the following sections:

- *About RVCT* on page 1-2
- *The ARM compiler and libraries* on page 1-3
- *Semihosting* on page 1-6.

1.1   About RVCT

RealView Compilation Tools (RVCT) consists of a suite of applications, together with supporting documentation and examples, that enable you to write applications for your ARM® architecture-based RISC processors. You can use RVCT to build C, C++, and ARM assembly language programs.

The RVCT toolkit consists of the following major components:
- command-line development tools
- utilities
- supporting software.

This book describes the ARM compiler and libraries provided with RVCT v2.0. See *ARM publications* on page x for a list of the other books in the RVDK documentation suite that give information on the assembler, linker, RealView Debugger, and other supporting software.

Where this document references the examples provided with RVCT v2.0, `Examples_directory` refers to `install_directory\RVCT\Examples\2.0\release\Windows`. If you accepted the default location during installation, `install_directory` refers to `C:\Program Files\ARM`.

1.2   The ARM compiler and libraries

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

1.2.1   The ARM compiler

The ARM compiler, `armcc`:

- Supports ISO C++ without exceptions and can generate ARM and Thumb instructions.

- Is tested against the Plum Hall C and C++ Validation Suite for ISO conformance, and compiles:
    — ISO C source into 32-bit ARM code
    — ISO C++ source into 32-bit ARM code
    — ISO C source into 16-bit Thumb code
    — ISO C++ source into 16-bit Thumb code.

- Is an optimizing compiler. Command-line options enable you to control the level of optimization.

- Generates output objects in *ARM Toolkit Proprietary ELF* (ATPE) format, and generates DWARF2 debug information. 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 ARM *Embedded Application Binary Interface* (EABI). See *ARM EABI compatibility* on page 2-3 for more details.

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

1.2.2   The C and C++ libraries

RVCT 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(). There are no target dependencies in the C++ library. The C++ libraries use the C libraries to provide target-specific support.

For more information on the Rogue Wave libraries, see the Rogue Wave HTML documentation and the Rogue Wave web site as 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 ISO C library for each combination of major build options, such as the ATPCS variant selected, the byte order of the target system, and the type of floating point support selected. See Chapter 5 *The C and C++ Libraries* for more information on the libraries.

1.2.3 Differences between the RVCT v2.0 compiler and previous ARM compilers

The RVCT v2.0 compiler now supports the following features:

- compliance with the ARM *Embedded Application Binary Interface* (EABI), see *ARM EABI compatibility* on page 2-3 for details

- double dashes -- to indicate command-line keywords, and single dashes - for command-line single-letter options (with or without arguments)

  ———— **Note** ————
  The single-dash options used in older versions of the ARM compiler are still supported for backwards-compatibility.

- embedded assembly as well as inline assembly code, see Chapter 4 *Inline and Embedded Assemblers* for details

- floating-point conformance using the --fpmode option, see *Defining optimization criteria* on page 2-37 for details

- the level of pointer alignment can be configured, see *Setting pointer alignment options* on page 2-42 for details

- the ARM architecture v6, and exploits the unaligned access behavior of ARMv6

- C++ template implicit inclusion, see *Template instantiation* on page 3-44 for details

---

- C++ namespaces, see *Namespaces* on page 3-45 for details

- control and manipulation of diagnostic messages, see *Diagnostic messages* on page 2-45 for details.

For other differences, see Appendix E *Older Compiler Options*.

1.3   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 RealView 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. The RVI-ME device for your MCU intercepts semihosting SWIs and requests service from the host computer.

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

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 2-8
- *Command syntax* on page 2-17.

2.1 About the ARM compiler

The ARM compiler, `armcc`, enables you to compile your ARM and Thumb® C and C++ code.

The RealView® Compilation Tools (RVCT) v2.0 compiler provides the functionality of the earlier ARM and Thumb C and C++ compilers in one executable.

The following sections are included:
- *Standards and compatibility*
- *Invoking the ARM compiler* on page 2-3
- *Source language modes* on page 2-6
- *Inline and embedded assembly language* on page 2-6
- *Library support* on page 2-7.

2.1.1 Standards and compatibility

By default, objects generated by the RVCT v2.0 C++ compiler are not binary-compatible with those generated by the older ARM C++ compilers. However, if you compile with the `--apcs /adsabi` option, the objects generated by the RVCT v2.0 C compiler are binary-compatible, with some restrictions, with those generated by the older ARM C compilers. See *EABI qualifier* on page 2-21 for more details.

———— **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 RVCT v2.0.

For backwards-compatibility, you can still use the old compiler names to invoke the new compiler in a particular mode (for example, `tcpp` for the Thumb C++ compiler). See *Invoking the ARM compiler using older tool names* on page 2-3 for details.

If you have older versions of the ARM compilers, see the documentation for that version for details.

For C language source code, the ARM compiler compiles C as defined by ISO/IEC 9899:1990 (E), including its Technical Corrigendums 1 and 2. Some features of C99 are also available, that is **long long** (see *long long* on page 3-26) and **restrict** (see *restrict* on page 3-27).

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

For C++ language source code, 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.

ARM EABI compatibility

RVCT v2.0 supports the ARM EABI. ARM EABI enables you to use ARM and Thumb objects and libraries from different producers that support EABI.

For more details of the ARM EABI, see the `ARM EABI home page` on the ARM DevZone™. You can access the ARM DevZone from `http://www.arm.com`.

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`.

Invoking the ARM compiler using older tool names

For backwards compatibility, you can still invoke the ARM compiler using one of the four tool names that were supported in the ARM compilation tools before RVCT v2.0. The startup configuration associated with each of the older tool names is shown in Table 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

By default the file suffix you specify changes the configuration assumed by the ARM compiler at start-up. 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++

Overriding the default behavior

The command-line options shown in Table 2-3 enable you to 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 2-3).

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, the following command-line causes the compiler to make determinations as shown in Table 2-4:

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

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

Table 2-4 Example configuration

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

The positioning of compiler options has no affect on the result. That is, in this example, placing the `--c90` option before the source filename gives the same result.

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.

2.1.3 Source language modes

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

ISO C In ISO C mode, the ARM compiler passes release 1999a of the *Plum Hall C Validation Suite* (CVS). This suite has been adopted by the British Standards Institute for C compiler validation in Europe. The compiler option `--strict` is used when running the tests.

C++ This mode applies only when compiling ARM C++. The ARM compiler compiles ISO/IEC standard C++. The compiler is tested against *Suite++, The Plum Hall Validation Suite for C++, version 00a*. This is the default language mode when compiling ARM C++. The option `--strict` was 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 2-23.

2.1.4 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*.

The following sections describe the inline and embedded assemblers:

- *Inline assembler* on page 4-2
- *Embedded assembler* on page 4-13
- *Differences between inline and embedded assembly code* on page 4-22.

2.1.5 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.

———— **Note** ————
The ARM C and C++ libraries comply with the ARM EABI. See *ARM EABI compatibility* on page 2-3 for more details.

———————————

2.2   File usage

This section describes naming conventions and included files.

The following sections are included:
- *Naming conventions*
- *Included files* on page 2-9
- *Precompiled header files* on page 2-11.

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 linking process. The names used on the command line, and as arguments to preprocessor #include directives, map directly to host file names under Windows/MS-DOS.

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

filename.c	ARM compiler recognizes .c, .cpp, .cp, .c++, .ac, .acpp, .tc, and .tcpp suffixes as source files.
filename.cc	Implicit include file. See *Implicit inclusion* on page 3-44 for more information on implicit inclusion.
filename.h	Header file (a convention only, this suffix has no special significance for the compiler).
filename.o	ARM object file in ATPE format.
filename.s	ARM or Thumb assembly language file. You can place this in the input file list. If you do, the compiler invokes the assembler, armasm, to assemble the file. Alternatively, if you specify the -S or --asm option, the ARM compiler outputs a file with this suffix, that has the same name as the input source file.
filename.lst	Error and warning list file (the default output extension for the --list option).
filename.txt	The default output extension when you use the -S or --asm option together with the -fs option.

Portability

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

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

- Make embedded pathnames 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 ATPE.

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 -fk and -fd compiler options
- the value of the environment variable RVCT20INC, 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 *includes_directory*\include and the compiler is seeking the include file sys\defs.h, it locates *includes_directory*\include\sys\defs.h if it exists.

When the compiler begins to process defs.h, the current place becomes *includes_directory*\include\sys. Any file included by defs.h that is not specified with an absolute pathname, is sought relative to *includes_directory*\include\sys.

The original current place *includes_directory*\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 -fk. 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 sought relative to the directory containing the source file that is being compiled.

The RVCT20INC environment variable

You can set the RVCT20INC environment variable points to the location of the included header and source files that are provided with RVCT v2.0. You must not change this environment variable. If you want to include files from other locations, then use the -I and -J command-line options as required. See *Include file options* on page 2-25 for more details.

When compiling from the command line, directories specified with RVCT20INC are searched immediately after directories specified by the -I option on the command line have been searched. If you use the -J option, RVCT20INC is ignored.

——— **Note** ———

If RVCT20INC is not set, then the ARMINC environment variable is used.

The search path

Table 2-5 shows how the various command-line options affect the search path used by the compiler when it searches for included header and source files. The following conventions are used in the table:

RVCT20INC   The list of directories specified by the RVCT20INC 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 2-10 for more information.

I*dirs* and J*dirs*

              The directories specified by the -I*dirs* and -J*dirs* compiler options.

Table 2-5 Include file search paths

Compiler option	\<include>	"include"
Neither -I nor -J	RVCT20INC	CP, RVCT20INC, and ARMINC
-J	J*dirs*	CP and J*dirs*
-I	I*dirs*, and RVCT20INC	CP, I*dirs*, RVCT20INC, and ARMINC
Both -I and -J	J*dirs* and I*dirs*	CP, I*dirs*, and J*dirs*
-fd	No effect	Removes CP from the search path, so the search is now the same as that invoked with angle brackets
-fk	No effect	Uses Kernighan and Ritchie search rules

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 the following sections:

- *Automatic PCH processing*
- *Manual PCH processing* on page 2-15
- *Controlling the output of messages during PCH processing* on page 2-16
- *Performance issues* on page 2-16.

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`. Unless you specify the `--pch_dir` option (see *PCH processing options* on page 2-25), it is created in the directory of the primary source file.

The header stop point

The PCH file contains a snapshot of all the code that precedes a *header stop point*. The header stop point is typically 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 the previous example, place it before `int`. See *Controlling PCH processing* on page 2-15 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 again `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 (see *Pragmas controlling PCH processing* on page 3-7) must not appear.

- 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 (that is, the one representing the most preprocessing directives from the primary source file) is used. 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).

This handles some common cases. Other PCH files must be deleted by you.

Manual PCH processing

You can specify the file name 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 file name and location*
- *Controlling PCH processing*.

Specifying a PCH file name and location

You can specify the file name 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 file name is appended to the directory name, unless the file name 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 2-25 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 3-7 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 2-12.

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, and gives the reason that it cannot be used.

Performance issues

Usually, 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. With sharing the disadvantage of large PCH files is minimized, 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 different requirements. However, be aware that making the best use of PCH support might require some experimentation and probably some minor changes to source code.

2.3   Command syntax

This section describes the command syntax for the ARM compiler.

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 -. For example, -J*directory*.

keyword options

All keyword options, or keyword options with arguments, are preceded by a double dash --. For example, --strict.

In most cases, the ARM compiler permits space between the option letter and the argument.

Wherever possible, the compiler adopts widely used command-line options familiar both to users of Windows/MS-DOS. The accepted compiler options are described in the rest of this section.

The following sections are included:

- *Invoking the ARM compiler* on page 2-18
- *Procedure Call Standard options* on page 2-20
- *Setting the source language* on page 2-23
- *Specifying search paths* on page 2-24
- *PCH processing options* on page 2-25
- *Setting preprocessor options* on page 2-26
- *C++ Language configuration and object generation* on page 2-28
- *Specifying output format* on page 2-30
- *Specifying the target processor or architecture* on page 2-33
- *Generating debug information* on page 2-35
- *Controlling code generation* on page 2-36
- *Diagnostic messages* on page 2-45.

2.3.1   Invoking the ARM compiler

The command for invoking the ARM compiler is:

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

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

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. For example, the --errors *efile* option is typically the first option specified (see *Diagnostic messages* on page 2-45).

The ARM compiler options are:

PCS-options
: This specifies the procedure call standard to use. See *Procedure Call Standard options* on page 2-20 for details.

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 Standard C++ when compiling C++ code. See *Setting the source language* on page 2-23 for details.

search-paths
: This specifies the directories that are searched for included files. See *Specifying search paths* on page 2-24 for details.

preprocessor-options

: This specifies preprocessor behavior, including preprocessor output and macro definitions. See *Setting preprocessor options* on page 2-26 for details.

C++-language
: This specifies the options specific to C++ compilation. See *C++ Language configuration and object generation* on page 2-28 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 2-30 for details.

target-options
: This specifies the target processor or architecture. See *Specifying the target processor or architecture* on page 2-33 for details.

debug-options This specifies whether or not debug tables are generated, and their format. See *Generating debug information* on page 2-35 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 2-36 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 2-45 for details.

warning-options This specifies whether specific warning messages are generated. See *Suppressing warning messages with the -W option* on page 2-48 for details.

additional-checks This specifies several additional checks that can be applied to your code, such as checks for data flow anomalies and unused declarations. See *Checking code with the -fa option* on page 2-48 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 2-47 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 .s extension) the compiler activates the ARM assembler to process the source file.

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, and use the following 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 by including, for example, --via *filename2* in the file. In the following example, the options specified in input.txt are read as the command-line is parsed:

```
armcpp --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. Input is terminated by entering `Ctrl-Z` then `Return`.

An assembly listing for the keyboard input is sent to the output stream after input has been terminated if both of 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.

Redirecting errors

Use the `--errors` *filename* option to redirect compiler error output to a file. Errors on the command line are not redirected.

2.3.2   Procedure Call Standard options

This section applies to the *ARM-Thumb Procedure Call Standard* (ATPCS) as used by the ARM compiler.

For more information on the ARM and Thumb procedure call standards, see the *ARM-Thumb Procedure Call Standard Specification* that is provided with RVDK.

See *Controlling code generation* on page 2-36 for other build options.

These command-line options enable you to specify the variant of the procedure call standard that is to be used by the compiler:

```
--apcs  qualifiers
```

The following rules apply to the `--apcs` command-line option:

• at least one qualifier must be present

- there must be no space between qualifiers.

If no --apcs or --cpu options are specified, the default is:

`--apcs /noswst/nointer/noropi/norwpi --fpu softvfp`

unless the default --fpu is overridden by the use of --cpu. See *Specifying the target processor or architecture* on page 2-33 for more information.

The qualifiers are listed in the following sections:
- *EABI qualifier*
- *Interworking qualifiers*
- *Position independence qualifiers* on page 2-22
- *Stack checking qualifiers* on page 2-23.

EABI qualifier

/adsabi   This option generates code that is compatible with the older ADS
              Application Binary Interface (ABI). In RVCT v2.0, **double** and **long long**
              data types are eight-byte aligned, which enables efficient use of the LDRD
              and STRD instructions in ARM architecture v5T 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 v2.0 C objects to be used with legacy C or C++
                objects

              - enable RVCT v2.0 C++ objects to be used with legacy C objects.

              ——— **Note** ———

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

              Legacy C++ objects are incompatible with RVCT v2.0 C++
              objects.

Interworking qualifiers

/nointerwork This option generates code with no ARM/Thumb interworking support.
              This is the default unless ARMv5T or later is specified by, for example,
              --cpu 5T, because ARMv5T or later provides direct interworking
              support.

/interwork   This option generates code with ARM/Thumb interworking support. See
             the *Using the Basic Linker Functionality* chapter in the *RealView
             Developer Kit v1.0 Linker and Utilities Guide* for information on the
             automatically generated interworking veneers. This is the default for
             ARMv5T or later.

Position independence qualifiers

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

/ropi   This option 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.

             ——— **Note** ———

             When compiling C++, polymorphic classes are not supported.

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

/rwpi   This option 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 be multiply instanced

                  —   data can be, but does not have to be, position-independent.

             •   Sets the PI attribute on read/write output sections.

             ——— **Note** ———

             Code that is not RWPI is automatically transformed into equivalent code
             that is RWPI. This is done by initializing pointers to writable data using
             a dynamic initialization.

Stack checking qualifiers

/noswstackcheck   This option uses the non-software-stack-checking ATPCS variant. This is the default.

/swstackcheck   This option uses the software-stack-checking ATPCS variant.

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 2-36).

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:

--old_cfe   Enables compilation using the legacy C compiler front-end. See Appendix E *Older Compiler Options* for more details of the older compiler options that are still supported.

--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 standard. Some minor extensions are also supported, for example // in comments and $ in identifiers.

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

--strict_warnings

--strict   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 you use. The --strict option causes errors to be issued, whereas the --strict_warnings option produces warnings. The error threshold is set so that the requested diagnostics are returned (see *Diagnostic messages* on page 2-45).

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 3-42 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. This is the default.
`armcc --strict`	Compiles strict ISO standard C.
`armcpp`	Compiles standard C++.
`armcpp --c90`	Compiles normal ISO standard C (C mode of C++).
`armcpp --c90 --strict`	
	Compiles strict ISO standard C (C mode of C++).
`armcpp --strict`	Compiles strict C++.

2.3.4 Specifying search paths

These options enable you to specify the directories to search for included files:
- *Include file options* on page 2-25 describes the options to search for include files
- *PCH processing options* on page 2-25 describes the options to control the processing of PCH files.

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 2-9 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 2-9 for complete details on how the compiler handles include files.

-fk   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 2-10 for more information. If you do not use this option, Berkeley-style searching is used.

-fd   Ensures that the handling of quoted include files is performed the same as for angle-bracketed include files. Specifically, the current place is excluded from the search path.

-J*directory*   Adds the specified *directory*, or comma-separated list of directories, to the end of the search path after all the directories specified by the -I options.

See *Included files* on page 2-9 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 2-11 for more details):

--pch   Automatically uses or creates a *PreCompiled Header* (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. This option can be used 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 options 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 2-30). 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 2-30 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 makefile 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 the -o *filename* option (see *Specifying output format* on page 2-30), 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*.

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

Defines a macro name as *def*. If = *def* is omitted, the compiler defines the name as 1. You can define function-style macros by appending a macro parameter list to *name*. There are no macro names defined by default (except for __LINE__ and similar language-mandated macros).

-D, *symbol=value*

Defines *symbol* as a preprocessor macro. This has the same effect as the text #define *symbol value* at the head of the source file. You can repeat this option.

-D*symbol*   Defines *symbol* as a preprocessor macro. This has the same effect as the text #define *symbol* at the head of the source file. You can repeat this option. The default value of *symbol* is 1.

-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.

2.3.7   **C++ Language configuration and object generation**

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

--dll_vtbl   Exports all virtual function tables automatically for exported classes. For more information, see the ARM EABI link of the ARM DevZone home page. You can access the ARM DevZone from http://www.arm.com/.

--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 3-44. 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 *Runtime Type Information* (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 3-45 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.

--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 3-44, 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. See *Template instantiation* on page 3-44, for more details.

>The default value is supplied by --dep_name.

>------ **Note** ------
>
>--dep_name cannot be combined with --no_parse_templates.

```
--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. For details on friend declarations, see *friend* on page 3-26. 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.

2.3.8   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.

--asm   Writes a listing of the assembly language generated by the compiler to a file (see the option -S if you do not want to generate object modules). Object code is generated and, unless the -c option is also used, the link phase is performed.

If used with -fs, the source code is interleaved with the assembly listing and output to a .txt file.

The output file names depend on the options used:

--asm   *inputname*.s is used for the resulting listing.

--asm -fs *inputname*.txt is used because the resulting interleaved code cannot be input to the assembler. See the -fs option.

--asm -c -o *newname.ext*

> There are two output files (usually *newname*.o for object code and *newname*.s for assembly). If *.ext* is not .s or .o, *newname.ext* is the name of the object file and *newname*.s is the name of the listing file.

--asm -fs -c -o *newname.ext*

> Gives the same output as --asm -c -o *newname.ext*, except that the listing file has interleaved source code and a .txt extension.

-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 that is described in *Setting preprocessor options* on page 2-26. (The -C option retains comments in preprocessor output.)

--list   Generates raw listing information in a file with a name based in the input filename. This information is typically 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.

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
C   An indication of a diagnostic, where:

R   Indicates a remark.

W   Indicates a warning.

E   Indicates an error.

C   Indicates a catastrophic error.

The line has the form:

`S "file-name" line-number column-number message-text`

where `S` can be `R`, `W`, `E`, or `C`. See *Severity of diagnostic messages* on page 2-46 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 catastrophic errors with an empty file name (`""`) and a line and column number of zero.

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

When a diagnostic 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, file name, line number, column number, and message text). However, the code letter is the lower-case 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 -S is assumed (unless -E is specified).
- Used with -c, it names the object file.
- Used with -S, it names the assembly language file.
- Used with -E, it specifies the output file for preprocessed source.
- If none of -c, -S, 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, and file1.s if -S is specified. If none of -c, -S, -E, or -o is present the default linker output name of __image.axf is used.

--depend *filename*

-MD   This option compiles the source and writes makefile dependency lines to a file. The output file is suitable for use by a make utility

If you use the `--depend` option, you can specify any file name. If you use the `-MD` option, the compiler names the file *filename*.d, where *filename* is the name of the source file.

-S   Writes a listing of the assembly language generated by the compiler to a file. However, unlike the `-asm` option, object modules are not generated. The name of the assembly output file defaults to *file*.s in the current directory, where *file* is the name of the source file stripped of any leading directory names. The default file name can be overridden with the `-o` option.

You can use `armasm` to assemble the output file and produce object code. The compiler adds `ASSERT` directives for command-line options such as ATPCS variants and byte order to ensure that compatible compiler and assembler options are used when reassembling the output. You must specify the same ATPCS settings to both the assembler and the compiler.

-fs   This option, when used with `-S` or `-asm`, interleaves C, or C++, source code line by line as comments within the compiler-generated assembler code. The output code is written to *file*.txt. A text file is output because the resulting interleaved code cannot be input to the assembler.

———— **Note** ————
If you use this option you cannot reassemble the output code listing from `-S`.

2.3.9   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** ————
RVDK does not support all ARM processors or architectures. See your Getting Started guide for the ARM processors and acrchitectures supported by RVDK.

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

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

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

`--thumb`   To target the Thumb instruction set.

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 4T` produces code that can be used by either the ARM7TDMI or ARM9TDMI®.

- If you specify a processor for the `--cpu` option, for example `--cpu ARM1020E`, 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 ARM7TDMI`.

- Specifying a Thumb-aware processor, such as `--cpu ARM7TDMI` 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` *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 ARM7TDMI. Wildcard characters are not accepted. Valid values are any ARMv6 or later ARM processor.

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

- Some `--cpu` selections imply an `--fpu` selection. For example, when compiling with the `--arm` option, `--cpu 4T` implies `--fpu softvfp`. The implicit FPU only overrides explicit `--fpu` options that appear before the `--cpu` option on the command line. If no `--fpu` option and no `--cpu` option are specified, `--fpu softvfp` is used.

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

3   ARMv3 without long multiply.

3M   ARMv3 with long multiply.

4   ARMv4 with long multiply but no Thumb.

4T	ARMv4 with long multiply and Thumb.
5T	ARMv5 with long multiply and Thumb.
5TE	ARMv5 with long multiply, Thumb, DSP multiply, and double-word instructions.
5TEJ	ARMv5 with long multiply, Thumb, DSP multiply, double-word instructions, and Jazelle extensions.
6	ARMv6 with long multiply, Thumb, DSP multiply, double-word instructions, unaligned and mixed-endian support, Jazelle extensions, and media extensions.

——— **Note** ———

See your Getting Started guide for the ARM processors and architectures supported by this version of RVDK.

--fpu *name* This option determines the target FPU architecture. If you specify this option, it only overrides any implicit FPU option that appears before the explicit --fpu option on the command line.

Valid options for *name* are:

none	Selects no floating-point option. No floating-point code is to be used.
softvfp	Selects software floating-point library (FPLib) with pure-endian doubles. This is the default if you do not specify a --fpu option.

2.3.10   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 2-37 for more information.

Debug table generation options

The following options specify how debug tables are generated:

-g   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.

Optimization options for debug code are specified by -0. By default, the -g option on its own is equivalent to:

-g -dwarf2 -O0 -gt+p

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

-gtp   This option, when used with -g, switches off the generation of debug table entries for preprocessor macro definitions. This reduces the size of the debug image. -gt-p is a synonym for -gtp.

-gt+p   This option, when used with -g, enables the generation of debug table entries for preprocessor macro definitions. This is the default option, and increases the size of the debug image. However, some debuggers ignore preprocessor entries.

Debug table format options

The following option specifies the format of the debug tables generated by the compiler:

--dwarf2   This option specifies DWARF2 debug table format. This is the default, and is the only available debug table format.

2.3.11 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:
- *Defining optimization criteria* on page 2-37
- *Controlling code and data sections* on page 2-41
- *Setting byte order* on page 2-42
- *Setting pointer alignment options* on page 2-42
- *Setting alignment options* on page 2-43
- *Controlling implementation details* on page 2-44.

Defining optimization criteria

These options control aspects of how the compiler optimizes generated code:

--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`.

--fpmode *model*

        Specifies the floating-point conformance, and sets library attributes and floating-point optimizations. *model* 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 run-time.

        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. It is not guaranteed that NaNs and infinities are produced in all circumstances defined by the IEEE model, or that when they are produced, they have the same sign. Also, it is not guaranteed that the sign of zero is that predicted by the IEEE model.

---

fast   Some value altering optimizations, where accuracy is sacrificed to fast execution. This is not IEEE compatible, and is not standard C.

This defines the symbol __FP_FAST.

--lower_ropi Enables less restrictive C and C++ in ROPI mode.

--lower_rwpi Enables less restrictive C and C++ in RWPI mode. This is the default.

--no_code_gen

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

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

-O0   Turns off all optimization, except some simple source transformations. This is the default optimization level if debug tables are generated with -g. It gives the best possible debug view and the lowest level of optimization.

-O1   Turns off optimizations that seriously degrade the debug view. If used with -g, this option gives a satisfactory debug view with good code density.

-O2   Generates fully optimized code. If used with -g, 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 if debug tables are not generated.

See *Pragmas* on page 3-2 for information on controlling optimization with pragmas.

-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.

--no_inlinemax

The compiler always attempts to inline, if possible.

-Ono_inline   Disables inlining of functions (see -Oinline). 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.

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

The compiler inlines functions as follows:

- Automatically, for optimization level 02, unless you use the option -Ono_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, because using the inline qualifier does not guarantee that functions are inlined. See *Function keywords* on page 3-8.

The compiler changes its criteria for inlining functions depending on whether you select -Ospace or -Otime. Selecting -Otime increases the likelihood that a function is inlined.

——— **Note** ———

When you set a breakpoint on an inline function, RealView Debugger attempts to set a breakpoint on each inlined instance of that function. If you are debugging an image in ROM, and the number of inline instances is greater than the number of available hardware breakpoints, RealView Debugger cannot set the additional breakpoints and reports an error.

_____

-Ono_autoinline

> Disables automatic inlining. This is the default for optimization levels -O1 and -O0.

-Oautoinline Enables automatic inlining. Automatic inlining is off by default for optimization levels -O0 and -O1, and on by default for optimization level -O2. 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.

-Ono_data_reorder

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

-Ono_ldrd   Disables optimizations specific to ARMv5TE and later processors.

-Oldrd   Enables optimizations specific to ARMv5TE and later processors. This is the default.

> If you select this option, and select an ARMv5TE or later --cpu option such as --cpu xscale, the compiler generates LDRD and STRD instructions where appropriate.

--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 STMs, and for LDMs that do not load the PC

- four, for LDMs that load the PC.

This option can reduce interrupt latency on ARM systems that:

- do not have a cache or a write buffer (for example, a cacheless ARM7TDMI)

- use zero-wait-state, 32-bit memory.

——— **Note** ———

Using this option increases code size and decreases performance slightly.

                   ARM DUI 0232B

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

This option has no significant benefit for cached systems, or for processors with a write buffer.

This option also 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. This is typically much greater than the latency introduced by multiple register transfers.

Controlling code and data sections

-zo   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. For example:

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

compiled with -zo gives:

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

This option increases code size sightly (typically by a few percent) for some functions because it reduces the potential for sharing addresses, data, and string literals between functions. However, this can help to reduce the final image size overall by enabling the linker to remove unused functions when you specify `armlink --remove`.

——— **Note** ———

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.

Linking object files created using the -zo option

When you link object files created with the -zo option specified, you can do the following:

• Specify the linker option -remove, to remove any unused functions from the final image.

- Use a scatter-loading description file to place some functions in fast memory and others in slow memory (see the *Using Scatter-loading description files* chapter in the *RealView Developer Kit v1.0 Linker and Utilities Guide*).

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

Setting byte order

--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.

--bigend This option generates code for an ARM processor using big-endian memory. With big-endian memory, the most significant byte of a word has the lowest address.

Setting pointer alignment options

--pointer_alignment=*num*

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

1 Accesses through pointers are treated as having an alignment of 1, that is, byte-aligned or unaligned.

2 Accesses through pointers are treated as having an alignment of at most 2, that is, at most halfword aligned.

4 Accesses through pointers are treated as having an alignment of at most 4, that is, at most word aligned.

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

De-aligning pointers might increase the code size, even on CPUs with unaligned access support, for example, on ARMv6, using the default 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 qualifer (see *Type qualifiers* on page 3-14).

Setting alignment options

`--memaccess` *option*

> This option indicates to the compiler that the memory in the target system has slightly restricted or expanded capabilities. By default, the ARM compiler assumes that the memory system can load and store words at four-byte alignment, halfwords at two-byte alignment, and bytes. Specify *option* to indicate the load and store capability:

`-UL41` Disables unaligned mode for ARMv6 processors. Unaligned support is used by default for packed data when you are compiling for ARMv6. That is, the compiler might generate unaligned word and halfword accesses, and might select a library that supports unaligned accesses.

`-S22` The memory cannot store halfwords. You can use this to suppress the generation of STRH instructions when generating ARM code for ARMv4 (and later) processors.

`-L22` The memory cannot load halfwords. You can use this to suppress the generation of LDRH instructions when generating ARM code for ARMv4 (and later) processors.

——— **Note** ———

Do not use `-L22` or `-S22` when compiling Thumb code.

It is possible that the processor has memory access modes available that the physical memory lacks (load aligned halfword, for example).

It is also possible that the physical memory has access modes that the processor cannot use (ARMv3 load aligned halfword, for example).

Controlling implementation details

-fy   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

This option is not recommended for general use and is not required for ISO-compatible source.

Code compiled with this option is not EABI compliant.

--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* (SJIS). 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.

-zc   Makes the **char** type to be signed. When **char** is signed, the macro __FEATURE_SIGNED_CHAR is defined by the compiler. The default is unsigned.

The sign of **char** is set by the last option specified that would normally affect it. For example, if you specify both --c90 and -zc options, and you want to make **char** signed, you must specify the -zc option *after* the --c90 option.

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

This option is not recommended for general use and is not required for ISO-compatible source. If used incorrectly, this option can cause errors in the resulting image.

Code compiled with this option is not EABI compliant.

2.3.12 Diagnostic messages

The compiler issues messages about potential portability problems and other hazards. The compiler options described in this section enable you 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, it is better to check the code than to switch off messages.

* Change the severity of specific messages.

This section describes:
* *Severity of diagnostic messages* on page 2-46
* *Controlling the output of diagnostic messages* on page 2-46
* *Changing the severity of diagnostic messages* on page 2-47
* *Suppressing diagnostic messages* on page 2-48
* *Suppressing warning messages with the -W option* on page 2-48
* *Checking code with the -fa option* on page 2-48
* *Suppressing error messages with the -Ec option* on page 2-49
* *Exit status codes and termination messages* on page 2-49.

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

If you have a previous version of the ARM compiler, then to help you to migrate your message options to the new interface, the legacy compiler options are still supported. See *Command Syntax Supported with --old_cfe* on page E-3 for a description of the options that are available. See *Mapping old compiler options to the new options* on page E-12 for the equivalent options to use in the RVCT v2.0 compiler.

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
Catastrophic error	Catastrophic errors indicate problems that cause the compilation to stop. These errors include command-line errors, internal errors, and missing include files. If multiple source files are being compiled, then no further source files are compiled. For example: `Serious error:  #5: could not open source file "myinclude.h"`
Error	Errors indicate violations in the syntactic or semantic rules of the C or C++ language. Compilation continues, but object code is not generated. 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 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 issued by default. Compilation continues, and object code is generated unless any 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`.

--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. If used, it is recommended that you specify this as the first option in the command line because any command-line errors for options preceding the --errors are written to stderr before redirection.

--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:

- New-style messages that can be changed have a -D postfix in the message number, for example:

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

- Old-style messages can also be changed, for example:

  ```
  Warning: C2068E: Uninitialised or corrupted use of PSR. This code may not
  work correctly
  ```

These options require a *tag*, which is the number of the message to be changed, and more than one tag can be specified. For example, to might want to change the following warning message to remark severity:

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

To do this, use the following option:

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

The following options are available:

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

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

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

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

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

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

Suppressing diagnostic messages

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

> This option disables all diagnostic messages that have the specified tag(s).
>
> This option requires a *tag*, which is the number of the message to be suppressed, and more than one tag can be specified. 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 produces old-style warning messages, as well as new-style messages. For example, the compiler might report the following warnings when compiling inline assembly code:
>
> ```
> Warning:  #1287-D: LDM/STM instruction may be expanded
> Warning: C2068E: Uninitialised or corrupted use of PSR. This code
> may not work correctly
> ```
>
> To suppress both of these messages, specify only the numerical part of the warning code, for example:
>
> ```
> armcc --diag_suppress 1287,2068 ...
> ```

Suppressing warning messages with the -W option

The `-W` option suppresses all warnings.

Checking code with the -fa option

The `-fa` option checks for certain types of data flow anomalies. The compiler performs data flow analysis as part of code generation. The checks indicate when an automatic variable might have been used before being assigned a value. The check is pessimistic and sometimes reports an anomaly where there is none. In general, it is useful at some stage to check all code using `-fa`.

Suppressing error messages with the -Ec option

The -Ec option suppresses all implicit cast errors, such as implicit casts of a nonzero **int** to **pointer**, for example:

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

Exit status codes and termination messages

If the compiler detected any remarks, warnings or errors during compilation, the compiler writes various messages to stderr:

- *filename*: *n* warning, *n* error indicates the detection of errors in the compilation (no message is written if no errors are detected). *n* indicates the number of warnings or errors detected.

- Compilation terminated. indicates that the compilation was prematurely terminated because of a catastrophic error.

- Compilation aborted. indicates that the compilation was prematurely terminated because of an internal error. This indicates an internal problem in the compiler and must be reported to your supplier.

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 2-46 for details on how the compiler handles the different levels of diagnostic messages.

Chapter 3
ARM Compiler Reference

This chapter gives information on features of the ARM® compiler that are specific to ARM Limited. It contains the following sections:

- *Compiler-specific features* on page 3-2
- *Language extensions* on page 3-19
- *C and C++ implementation details* on page 3-32
- *Predefined macros* on page 3-48.

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 aspects of the ARM compiler that are specific to ARM Limited, including:

- *Pragmas*
- *Function keywords* on page 3-8
- *Variable declaration keywords* on page 3-11.

——— **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
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
check_printf_formats	Off	*Pragmas controlling printf and scanf argument checking* on page 3-3
check_scanf_formats	Off	*Pragmas controlling printf and scanf argument checking* on page 3-3
check_stack	On	*Pragmas controlling code generation* on page 3-5
debug	On	*Pragmas controlling debugging* on page 3-4

Table 3-1 Pragmas recognized by the ARM compiler (continued)

Pragma name	Default	Reference
hdrstop	–	*Pragmas controlling PCH processing* on page 3-7
import	–	*Pragmas controlling code generation* on page 3-5
no_pch	–	*Pragmas controlling PCH processing* on page 3-7
once	–	*Pragmas controlling code generation* on page 3-5
Ospace	–	*Pragmas controlling optimization* on page 3-4
Otime	–	*Pragmas controlling optimization* on page 3-4
O*num*	–	*Pragmas controlling optimization* on page 3-4
pop	–	*Pragmas for saving and restoring the pragma state*
push	–	*Pragmas for saving and restoring the pragma state*
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

This pragma 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

> This pragma 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
> This pragma 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 optimization

The pragmas that control optimization must be placed outside of a function because they work at a per-function level. That is, you cannot apply more than one optimization level on a function. The following pragmas control aspects of optimization:

Ospace
> This pragma optimizes for space.

Otime
> This pragma optimizes for time.

O*num*
> This pragma changes optimization level. The value of *num* is 0, 1, or 2. See *Defining optimization criteria* on page 2-37 for more information on optimization levels.

Pragmas controlling code generation

The following pragmas control how code is generated (other code generation options are available from the compiler command line):

arm   This pragma switches code generation to the ARM instruction set.

thumb   This pragma switches code generation to the Thumb® instruction set.

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

check_stack   This pragma reenables 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.

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

This pragma 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 3-8). The pragma form can be useful when applied to an entire interface specification (header file) without altering that file.

import(*symbol_name*)

>   This pragma 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. For an example of its use, see *Avoiding the semihosting SWI* on page 5-10.

arm section *section_sort_list*

>   This pragma 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
>   - rwdata
>   - rodata
>   - 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 bar

#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 .constdata

#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 linker to control placing a named section at a particular address in memory (see the *Using Scatter-loading description file* chapter in the *RealView Developer Kit v1.0 Linker and Utilities Guide*).

Pragmas controlling PCH processing

The following pragma control PCH processing:

hdrstop   This pragma 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   This pragma suppresses PCH processing for a given source file.

See *Precompiled header files* on page 2-11 for a detailed description of PCH files.

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 specification, and are grouped in the following sections:

- *Declarations inside functions*
- *Function qualifiers*
- *Function storage class modifiers* on page 3-11.

Declarations inside functions

Declarations inside a function indicate that the following statements are processed differently. The **asm** keyword does not modify the surrounding function, but it does indicate that the statements following the keyword are different.

```
__asm("assembler-instruction[;assembler-instruction]");
__asm{assembler-code}
```

> Instructs the compiler that the statement is written in assembler language. See *Inline assembler syntax* on page 4-2 for more details on inline assembler. _asm is a synonym for __asm. The **asm** keyword is not accepted in C++.

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.

```
__declspec(dllexport)
```

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

__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 ATPCS. The default ATPCS mode must be used. The function exits by setting the pc to lr–4 and the CPSR to the value in SPSR. It is not available when compiling for Thumb. No arguments or return values can be used with __irq functions.

__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 ATPCS 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 ATPCS.

`__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:

`swi_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.

`__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.

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 4-13 for more details on embedded assembler. `_asm` is a synonym for `__asm`. The **asm** keyword is not accepted in C++.

`__forceinline`

> Forces the compiler to compile a 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 2-37 for information on command-line options that affect inlining.

`__inline`   Instructs the compiler to compile a 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 2-37 for information on command-line options that affect inlining.

3.1.3   Variable declaration keywords

This section describes the implementation of various standard and ARM-specific variable declaration keywords. Standard C or C++ keywords that do not have ARM-specific behavior or restrictions are not documented. See also *Type qualifiers* on page 3-14 for information on qualifiers such as **volatile** and `__packed`.

The keywords are grouped in the following sections:

- *Standard keywords*
- *ARM-specific keywords*
- *Type qualifiers* on page 3-14.

Standard keywords

These keywords declare a storage class.

`register`   Using the ARM compiler, you can declare any number of local objects (auto variables) to have the storage class `register`.

> ──── **Note** ────
>
> Using `register` is not recommended because the compiler is very effective at optimizing code. The `register` keyword is regarded by the compiler as a suggestion only. Other variables, not declared with the `register` keyword, can be kept in registers and register variables can be kept in memory. Using `register` might increase code size because the compiler is restricted in its use of registers for optimization.

Depending on the variant of the ATPCS being used, there are between five and seven integer registers available, and four floating-point registers. In general, declaring more than four integer register variables and two floating-point register variables is not recommended.

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.

ARM-specific keywords

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 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 ATPCS 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 ATPCS 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 ATPCS variant used, between five and seven integer registers, and four floating-point registers are available for use as global register variables. In practice, using more than three global integer register variables in ARM code, or one global integer register variable in Thumb code, or more than two global floating-point register variables is *not* recommended.

——— **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.

Type qualifiers

This section describes the implementation of various standard C and ARM-specific type qualifiers. 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 ARM-specific behavior or restrictions are not documented:

__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. 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(8) 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.

——— **Note** ———

The *ARM-Thumb Procedure Call Standard* 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.

__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 speaking, 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 3-39.

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 (Example 3-2 on page 3-16).

Example 3-2 Pointer to packed

```
__packed int *p
```

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 (Example 3-3).

Example 3-3 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. 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.

__weak   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 effectively makes the branch a no-op:

```
__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 code below 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 (Example 3-4).

Example 3-4 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 *Creating and Using Libraries* chapter in the *RealView Developer Kit v1.0 Linker and Utilities Guide* for details on library searching.

3.2   Language extensions

This section describes the language extensions supported by the ARM compiler, and includes the following sections:

*   *C language extensions*
*   *C and C++ language extensions* on page 3-25.

3.2.1   C language extensions

The compiler supports the ISO C language extensions described below and in *C and C++ language extensions* on page 3-25. The extensions are not available if the compiler is restricted to compiling strict ISO C, for example, by specifying the -strict compiler option.

// comments

The character sequence // starts a 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, such as the following, are supported in initializers:

```
int i;
int j = (int)&i; /* but not allowed by ISO */
```

__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.

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 the redundancy is ignored. For example, **short short** or **unsigned unsigned**. The following error is issued:

  ```
  #240: duplicate specifier in declaration
  ```

Floating-point extensions

The following floating-point extensions are supported:

- **long float** is accepted as a synonym for **double**.

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.

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 might 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)

  — among the restrictions, the extension only applies to constructs within a **struct**.

- 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
```

- Bit fields can have base types that are **enum** types or integral types besides **int** and **unsigned int**.

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 name space 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.)

Other C language extensions

The following extensions are also supported:

- An input file can contain no declarations. For C and C++, a remark is issued.

    If you specify the `--strict_warnings` option, this remark is upgraded to a warning.

    If you specify the `--strict` option, this remark is upgraded to an error.

- 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 or error 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 warning is issued in strict ISO mode:

    `#301: typedef name has already been declared (with same type)`

- 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.

- 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 ISO C mode:

  ```
  #64: declaration does not declare anything
  ```

3.2.2 C and C++ language extensions

This section describes the extensions to both the ISO C language, and the ISO/IEC C++ language that are accepted by the compiler. See *C language extensions* on page 3-19 for those extensions that apply only to C. None of these extensions are available if the compiler is restricted to compiling strict ISO C or strict ISO/IEC C++. This is the case, for example, when the -strict compiler option is specified.

Except where noted, all of the extensions described in *C language extensions* on page 3-19 are also enabled in C++ mode.

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.

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

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 ISO 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"
};
```

restrict

`restrict` can be used as a type qualifier for object pointer types and function parameter arrays that decay to pointers. The presence of this keyword is recorded to ensure that the compiler can perform optimizations that would otherwise be prevented because of possible aliasing. The keyword is enabled by --restrict. This is also available in C++.

If you use the --restrict command-line option to recognize that the `restrict` keyword is enabled, the C99 `restrict` feature is supported in a form extended for C++. This enables `restrict` as a type qualifier for reference and pointer-to-member types and for non-static member functions. The set of C++ extensions is described in J16/92-0057.

Inline and embedded assembler extensions

The following inline and embedded assembler extensions are supported:

* `asm` statements and declarations are accepted, see *Inline assembler* on page 4-2. The inline assembler is a way of extending the compiler, but there are limitations.

* `asm` functions are designated as embedded assembler, see *Embedded assembler* on page 4-13. `__asm` is a synonym for `asm`. 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.

nop() instrinsic

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. One NOP instruction is generated for each `__nop()` in the source. The compiler does not optimize-away NOPs, except for normal dead-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.

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;
}
```

Keywords

ARM implements some keyword extensions for functions and variables. See:

*   *Function keywords* on page 3-8
*   *Variable declaration keywords* on page 3-11
*   *Type qualifiers* on page 3-14.

Hexadecimal floating-point constants

ARM 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**.

0d_*nn*   Interpret a 16-digit hex number *nn* as a **double**.

There must be exactly eight digits for **float** constants. There must be exactly 16 digits for **double** constants.

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 /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 /rwpi option deviates from the C++ standard. The declarations in Example 3-5 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.

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, unions and structs

An extension is supported to enable an anonymous union to be introduced into a
containing class by a **typedef** name. It does not have be declared directly, as with a true
anonymous union. For example:

```
typedef union {
  int i, j;
} U;             // U identifies a reusable anonymous union.
class A {
  U;             // Okay -- references to A::i and A::j are allowed.
};
```

In addition, the extension also enables anonymous classes and anonymous structs, 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, structs, or unions. For example,

```
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. Here's an 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";
```

Preprocessing

The preprocessing symbol c_plusplus is defined in addition to the standard __cplusplus.

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.

3.3   C and C++ implementation details

This section describes implementation details for the ARM compiler, including:

- *Character sets and identifiers*
- *Basic data types* on page 3-34
- *Operations on basic data types* on page 3-36
- *Structures, unions, enumerations, and bitfields* on page 3-37
- *Change to ::operator new function* on page 3-42
- *Tentative arrays not supported* on page 3-42
- *Old-style C parameters in C++ functions* on page 3-42
- *Anachronisms* on page 3-42
- *Template instantiation* on page 3-44
- *Namespaces* on page 3-45
- *Extern inline function* on page 3-47.

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 5-25.)

- 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.

- 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.

- 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 -zc option can be used to make the **char** signed.

- 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.

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

Escape sequence	Char value	Description
\r	13	Carriage return
\xnn	0xnn	ASCII code in hexadecimal
\nnn	0nnn	ASCII code in octal

3.3.2   Basic data types

This section gives information about how the basic data types are implemented in ARM C and C++.

———— **Note** ————

You can more information on basic data types on the ARM EABI link of the ARM DevZone home page, http://www.armdevzone.com/.

Size and alignment of basic data types

Table 3-3 gives the size and natural alignment of the basic data types. Type alignment varies according to the context. (See *Structures, unions, enumerations, and bitfields* on page 3-37.)

———— **Note** ————

If you specify the command-line option --apcs /adsabi, the alignment for the **double** and **long long** types is 4.

• 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.

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)

Table 3-3 Size and alignment of data types (continued)

Type	Size in bits	Natural alignment in bytes
`long`	32	4 (word-aligned)
`long long`	64	8 (doubleword-aligned)
`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)

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.

If softvfp is selected, 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 3-37 for more information.

ARM implements an ISO extension for floating-point constants (see *Hexadecimal floating-point constants* on page 3-28).

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 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 5-18 on page 5-94.

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.

——— **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 5-51 and Chapter 6 *Floating-point Support* 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 obtained as if by the expression:

  `((int)a - (int)b) / (int)sizeof(type pointed to)`

- If the pointers point to objects whose size is 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.

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)
- **signed int**.

Implementing **enum** in this way can reduce data size. The command-line option -fy sets the underlying type of **enum** to **signed int**. See *About the ARM compiler* on page 2-2 for more information on the -fy option.

Unless you use the -strict option, **enum** declarations can have a comma at the end as in:

```
enum { x = 1, };
```

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 3-42 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. Figure 3-1 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 two 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 (Figure 3-1).

- Use the -W+s option to generate a warning when the compiler inserts padding in a **struct**.

- Structures with empty initializers are permitted in C++ and only warned about in C (if C and -strict an error is generated):

    struct { int x; } X = { };

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 3-14. 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.

A bitfield container can be any of the integral types.

——— **Note** ———

The compiler warns about non `int` bitfields. You can disable this warning with the `-Wb` compiler option.

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 above:

```
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 above example, 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 (Figure 3-2).

```
struct X {
    int x:10;
    char y:8;
};
```
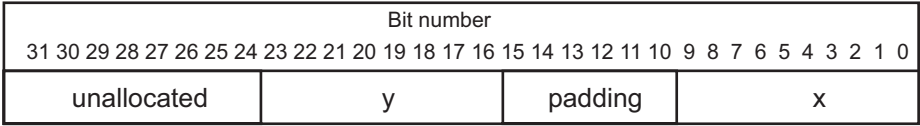

Figure 3-2 Bitfield allocation 1

—— **Note** ——

The same basic rules apply to bitfield declarations with different container types. For example, adding an **int** bitfield to the example above 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 (Figure 3-3 on page 3-42).

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 Change to ::operator new function

When the function `::operator new` function runs out of memory, it now raises the signal `SIGOUTOFHEAP`, instead of returning `NULL`. See *Signal function* on page 5-94 for more details.

3.3.6 Tentative arrays not supported

The legacy C++ compilers enabled you to use the extension `int a[]`. You cannot use tentative arrays when compiling C++ with the RVCT v2.0 compiler.

3.3.7 Old-style C parameters in C++ functions

The legacy C++ compilers enabled you to use old-style C parameters in C++ functions. That is,

```
void f(x) int x; { }
```

This is only supported in the RVCT v2.0 compiler if you use the `--anachronisms` compiler option.

3.3.8 Anachronisms

The following anachronisms are accepted when you enable anachronisms with the `--anachronisms` option (see *Setting the source language* on page 2-23):

- **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 nonnested 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.

———————————

3.3.9   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.

See *C++ Language configuration and object generation* on page 2-28 for details of the 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 2-28 for details of these options.

Implicit inclusions are only performed during the normal compilation of a file, (that is, when not using the --preproc_only command-line option). A common method of investigating certain kinds of problems is to produce a preprocessed source file that can be inspected. When using implicit inclusion it is sometimes desirable for the preprocessed source file to include any implicitly included files. This can be done using the --no_preproc_only command-line option, and causes the preprocessed output to be generated as part of a normal compilation. When implicit inclusion is being used, the implicitly included files appear as part of the preprocessed output in the same location that they were included in the compilation. See *Setting the source language* on page 2-23 for details of these options.

3.3.10 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 2-28 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);
```

3.3.11 Extern inline function

The 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 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 can not be inlined. For example, a recursive function can not be inlined:

  ```
  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 3-11), for example:

  ```
  __forceinline int g() {return 1;}
  ```

3.4   Predefined macros

Table 3-4 lists the macro names predefined by the ARM compiler for C and C++. Where the value field is empty, the symbol is only defined.

Table 3-4 Predefined macros

Name	Value	When defined
__arm	–	Always defined for the ARM compiler. ——— **Note** ——— This macro is also defined if you invoke the ARM compiler using the older commands, `armcpp`, `tcc`, and `tcpp`.
__ARMCC_VERSION	*ver*	For giving the version number of the compiler. It is a decimal number, whose value can be relied on to increase between releases. The format is *PVtbbb* where: *P* is the major version (2 for RVCT v2.0) *V* is the minor version (0 for RVCT v2.0) *t* is the patch release (0 for RVCT v2.0, unpatched) *bbb* is the build (841 for example). The example given results in 200841.
__APCS_ADSABI	–	If you use the --apcs /adsabi command-line option.
__APCS_INTERWORK	–	If you use the --apcs /interwork command-line option or set the cpu architecture to 5TE.
__APCS_ROPI	–	If you use the --apcs /ropi command-line option.
__APCS_RWPI	–	If you use the --apcs /rwpi command-line option.
__APCS_SWST	–	If you use the --apcs /swst command-line option.
__ARRAY_OPERATORS	–	In C++ compiler mode when array new and delete are enabled.
__BIG_ENDIAN	–	If compiling for a big-endian target.
_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.

Table 3-4 Predefined macros (continued)

Name	Value	When defined
`__CC_ARM`	1	Always set to 1 for the ARM compiler.
`__DATE__`	*date*	When date of translation of source file is required.
`__EDG__`	–	Always defined, unless you use the `--old_cfe` command-line option.
`__EDG_IMPLICIT_USING_STD`	–	In C++ compiler mode when `implicit_using_std` is set to `TRUE` by a command-line option.
`__EDG_VERSION__`	–	Always set to an integer value that represents the version number of the EDG front-end, unless you use the `--old_cfe` command-line option. For example, version 3.0 is represented as 300.
`__FEATURE_SIGNED_CHAR`	–	Set by `--signed` or `-zc` (used by `CHAR_MIN` and `CHAR_MAX`).
`__FILE__`	*name*	The presumed full pathname of the current source file.
`__func__`	*name*	The name of the current function.
`__FP_FAST`	–	When the `--fpmode fast` option is specified.
`__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.
`__FP_INEXACT_EXCEPTION`	–	When you specify the `--fpmode ieee_full` option.
`__IMPLICIT_INCLUDE`	–	When you specify the `--implicit_include` option.
`__LINE__`	*num*	When line number of the current source file is required.
`__MODULE__`	*mod*	Contains the filename part of the value of `__FILE__`.
`__OPTIMISE_LEVEL`	–	If optimize level is set to 0, 1 or 2.
`__OPTIMISE_SPACE`	–	If `-Ospace` in use.
`__OPTIMISE_TIME`	–	If `-Otime` in use.

Table 3-4 Predefined macros (continued)

Name	Value	When defined
`__PLACEMENT_DELETE`	–	In C++ compiler mode when placement delete is enabled. This is the default.
`__prettyfunc__`	*name*	The unmangled name of the current function.
`__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 library. Set if using `--fpu softvfp` for ARM or Thumb.
`__STDC__`	–	In all compiler modes.
`__STDC_VERSION__`	–	Standard version information.
`__STRICT_ANSI__`	–	If you use the `--strict` command-line option.
`__TARGET_ARCH_`*xx*	–	*xx* represents the target architecture and its value depends on the target architecture. For example, if the compiler options `--cpu 4T` or `--cpu ARM7TDMI` are specified then `__TARGET_ARCH_4T` 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 the compiler option `--cpu ARM7TM` is specified then `__TARGET_CPU_ARM7TM` is defined and no other symbol starting with `__TARGET_CPU_` is defined. If the target architecture is specified, then `__TARGET_CPU_generic` is defined. If the processor name contains hyphen (-) characters, these are mapped to an underscore (_). For example, `--cpu SA-110` is mapped to `__TARGET_CPU_SA_110`.
`__TARGET_FEATURE_DOUBLEWORD`	–	If the target architecture supports the `PLD`, `LDRD`, `STRD`, `MCRR`, and `MRRC` instructions. That is, ARMv5T and later.

Name	Value	When defined
`__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 is Thumb-capable (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 2-33 for more information on FPU options.
`__thumb`	–	If you use the `--thumb` command-line option. ——— **Note** ——— This macro is also defined if you invoke the compiler in Thumb mode using the older commands `tcc`, and `tcpp`.
`__TIME__`	*time*	When time of translation of the source file is required.
`_WCHAR_T`	–	In C++ compiler mode, to specify that **wchar_t** is a keyword.

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 4-13
- *Differences between inline and embedded assembly code* on page 4-22.

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++.

The following sections are included:

- *Inline assembler syntax*
- *Virtual registers* on page 4-4
- *Instructions* on page 4-5
- *Operands* on page 4-6
- *Function calls and branches* on page 4-8
- *Labels* on page 4-10
- *Differences from previous versions of the ARM C/C++ compilers* on page 4-10
- *Restrictions on inline assembly operations* on page 4-11.

For more information, see *RealView Developer Kit v1.0 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
- *Inline assembler example* on page 4-3.

You can use an `asm` or `__asm` statement anywhere a statement is expected.

The inline assembler supports the full ARM instruction set, including generic coprocessor instructions, but not `BX` and `BLX`.

Inline assembly with the asm keyword

The ARM compiler supports the syntax in the ISO/IEC C++ standard, with the restriction that the string-literal must be a single string, for example:

```
asm("instruction[;instruction]");
```

The **asm** declaration must be inside a C++ function. You cannot include comments in the string literal.

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. Inline assembler code using the assembler specifier has the following format:

```
__asm
{
    instruction [; instruction]
    ...
    [instruction]
}
```

If two instructions are on the same line, you must separate them with a semicolon. If an instruction requires more than one line, line continuation must be specified with the backslash character \. You can use C or C++ comments anywhere within an inline assembly language block.

Inline assembler example

Example 4-1 shows inline assembly code that copies a string value. This code is also in *Examples_directory*\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;
}
```

4.1.2   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 pc (r15), lr (r14), and sp (r13) registers, and they cannot be read or directly modified in inline assembly code. Any attempt to use these registers results in the error message. For example, if r14 is specified:

```
#20: identifier "r14" is undefined
```

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.

A warning is given the first time a virtual or physical register name is used, and only once for each translation unit, unless you have explicitly declared the name in the C or C++ code. It is better to use C or C++ variables as instruction operands. For example, if you specify register r3, the following warning is displayed:

```
#1267-D: Implicit physical register R3 should be defined as a variable
```

4.1.3   Instructions

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. Examples of instructions that might be replaced by equivalent ARM instructions are LDM, STM, LDRD, and STRD.

The ARM compiler outputs a message informing you it might expand instructions, for example:

```
#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.

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.

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* on page 4-6) 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.

4.1.4   Operands

Operands can be one of the following types:

* *Virtual registers*
* *Expression operands*
* *Register lists* on page 4-7
* *Intermediate operands* on page 4-8.

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 4-4 for more details.

——— **Note** ———

The initial value in each virtual register is unpredictable. You must write to virtual registers before reading from them. The compiler gives a warning if you attempt to read a virtual register before writing to it.

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, excluding `long long`, or a pointer type. The expression must be a modifiable l-value 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 that is 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.

4.1.5 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*
- *Input parameter list* on page 4-9
- *Output value list* on page 4-9
- *Corrupted register list* on page 4-9.

—— **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`, 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` 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, `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 l-value 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, `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`, 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.

4.1.6 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 for branch instructions in inline assembly code.

4.1.7 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 a legacy ARM compiler. If you have an older version of the ARM compiler, see the documentation for that ARM compiler for more details.

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 a warning because it reads virtual registers r0 and r1 before writing to them. 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 now 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 previous behavior where the lowest numbered register always references 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.

4.1.8 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.

Registers

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.

Processor modes

It is possible to change the processor mode using inline assembly code. However, changing processor mode inhibits the use of C or C++ operands, or calls to compiled C or C++ code, until the processor mode is changed back.

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.

Vector Floating-Point coprocessor (VFP) and Floating-Point Accelerator (FPA)

The inline assembler does not provide direct support for VFP or FPA instructions although you can specify them using the generic coprocessor instructions.

Inline assembly code must not be used to change VFP vector mode. Inline assembly can contain floating point expression operands that can be evaluated using compiler-generated VFP code. Therefore, it is important that the state of the VFP is modified only by the compiler.

Unsupported instructions

The instructions BX, BXJ, BLX, and BKPT are not supported in the inline assembler.

Transformable instructions

The instructions LDM, STM, LDRD, and STRD might be transformed into a sequence of equivalent ARM LDR or STR instructions.

 ARM DUI 0232B

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 4-15
- *Differences between expressions in embedded assembler and C or C++* on page 4-15
- *Generation of embedded assembly functions* on page 4-16
- *The __cpp keyword* on page 4-18
- *Manual overload resolution* on page 4-18
- *Keywords for related base classes* on page 4-19
- *Keywords for member function classes* on page 4-19
- *Calling non-static member functions* on page 4-20.

For more information, see the *RealView Developer Kit v1.0 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` function qualifier, and can be used on:
- member functions
- non-member functions
- template functions
- template class member functions.

Functions declared with `__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]
}
```

─── **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 4-2 is equivalent to Example 4-1 on page 4-3, but modified to show how to use the string copy routine as an embedded assembler routine.

Example 4-2 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 4-19 and *Keywords for member function classes* on page 4-19):

```
__cpp(expr)
__offsetof_base(D, B)
__mcall_is_virtual(D, f)
__mcall_is_in_vbase(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 4-16).

  ————————————

- `__asm` functions do not change the ATPCS rules that apply. This means that all calls between an `__asm` function and a normal C or C++ function must adhere to the ATPCS, 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 assembler and C or C++

Be aware of the following differences between embedded assembler 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 zeroes 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 4-18 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.

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 the area with the name `.emb_text`. 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 inline 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 `.asm` 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)
    MOV pc, lr
}
```

4.2.7   Keywords for related base classes

The following keywords enable 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
>     MOV pc, 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.

`__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_vbaseptr(D, f)`, otherwise it must be done with `__mcall_this_offset(D, f)`.

`__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 will be skipped. See the general example below.

`__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)
  ASSERT {FALSE}   // can't access virtual base
 else
  MOV r0, rp                                  // set up this pointer for D*
  ADD r0, r0, #__mcall_this_offset(D, f)      // do this adjustment
 endif
  BL __cpp(&D::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)
  ASSERT {FALSE}   // can't access virtual base
 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()
```

4.3 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
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 4-4)
Return instructions	You must add them in your code.	Generated automatically

——— **Note** ———

A list of differences between embedded assembler and C or C++ is provided in *Differences between expressions in embedded assembler and C or C++* on page 4-15.

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:

5.1   About the runtime libraries

The following runtime libraries are provided to support compiled C and C++:

ISO C   The C libraries consist of:

- The functions defined by the ISO C library standard.

- Target-dependent functions used to implement the C library functions in the semihosted execution environment. You can redefine these functions in your own application.

- Helper functions 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.

For a detailed description of how the libraries comply with the ISO standard, see *ISO implementation definition* on page 5-91.

As supplied, the ISO C libraries use the standard ARM semihosted environment to provide facilities such as file input/output. This environment is supported by your RealView ICE Micro Edition (RVI-ME) device. You can use the ARM development tools in RealView Compilation Tools (RVCT) to build applications. To debug the applications under on your development board, use RealView Debugger. See the description of semihosting in Chapter 7 *Semihosting* for more information on the debug environment.

———— **Note** ————

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.

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.

5.1.1   Build options and library variants

When you build your application, you must make certain fundamental choices. For
example:

Byte order   Big-endian or little-endian.

Floating-point support

> Software or none.

Stack limit   Checked or unchecked.

Position-independence

> Data can be read/write position-independent or not position-independent.
> Code can be read-only position-independent or not position-independent.

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 *Creating and Using Libraries* chapter in the *RealView Developer Kit v1.0
    Linker and Utilities Guide*

*   *Procedure Call Standard options* on page 2-20 for the compiler

*   the *RealView Developer Kit v1.0 Assembler Guide* for the assembler.

5.1.2 Library directory structure

The libraries are installed in two subdirectories within *includes_directory*\lib:

armlib   This subdirectory contains the variants of the ARM C library, the
            floating-point arithmetic library, and the math library. The accompanying
            header files are in *includes_directory*\include.

cpplib   This subdirectory contains the variants of the Rogue Wave C++ library
            and supporting C++ functions. The Rogue Wave and supporting C++
            functions are collectively referred to as the ARM C++ Libraries. The
            accompanying header files are installed in *includes_directory*\include.

The environment variable RVCT20LIB 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 licence fee. See the Rogue Wave online documentation in
  includes_directory\Html for more about the C++ library.

5.1.3 Reentrancy and static data

Libraries that make use of static data are supplied in two variants:

• Static data addressed in a position-dependent fashion. Code from these variants is
  single threaded. Library c_a__un, for example, has position-dependent data.

• Static data addressed in a position-independent fashion using offsets from the
  static base register sb (r9). Code from these variants can be multiply-threaded and
  is reentrant. Library c_a__ue, for example, has position-independent data.

The following points describe how static data is used by the libraries:

• Floating-point arithmetic libraries do not use static data and are always reentrant.

• All statically-initialized data in the C libraries is read-only.

• All writable static data is uninitialized.

• Most C library functions use no writable static data and are reentrant whether built with base build options (-apcs /norwpi) or reentrant (-apcs /rwpi) build options.

• A few functions have static data in their definitions (see Table 5-1). You must not use these, or other similar functions, in a reentrant application unless you build it -apcs /rwpi.

Table 5-1 Library functions that use static data

Function	Description
strtok()	Contains implicit static data
gamma() and lgamma()	These functions, in math.h, use a global variable called signgam
rand() and srand()	Require a random seed
stdin, stdout, and stderr	These are static data
atexit()	Stores exit handlers in static data
setlocale(), asctime(), localtime(), localeconv(), and tmpnam()	Return pointers to static data
__user_libspace()	This function is used by many other routines
_sys_clock()	The default implementation has a static variable that stores the time-at-start-of-program
fenv.h functions	These are used to install FP exception traps
signal.h functions	These are used to install signal handlers

——— **Caution** ———

The number of functions that use static data in their definitions might change in future versions of RVCT.

———————————

5.2 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.

There are three major 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 non-hosted application that can, for example, be embedded into ROM. See *Building an application for a nonsemihosted environment* on page 5-7.

- Build an application that does not use `main()` and does not initialize the library. This application has, unless you re-implement some functions, restricted library functionality. See *Building an application without the C library* on page 5-12.

5.2.1 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 your RVI-ME

- implementing your own SWI handler for the semihosting SWI (see Chapter 7 *Semihosting*).

A list of functions that require semihosting is given in *Overview of semihosting dependencies* on page 5-8.

You are not required to write any new functions or include files if you are using the default semihosting functionality of the library.

Using RVI-ME

The ARM debug agents support 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 5-67.

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* 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 non-hosted environment by one of the following methods:

- Removing all calls to semihosted functions. See *Avoiding the semihosting SWI* on page 5-10.

- Re-implementing the semihosted functions. See *Building an application for a nonsemihosted environment*. 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 is used in ARM state and 0xab is used in Thumb state. See the semihosting SWI definitions in Chapter 7 *Semihosting*. See also the include file rt_sys.h for definitions of the functions that call the semihosting SWI.

5.2.2 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 5-10.

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.

The functions that you might have to re-implement are described in:

•   *Tailoring static data access* on page 5-24

•   *Tailoring locale and CTYPE* on page 5-25

•   *Tailoring error signaling, error handling, and program exit* on page 5-51

•   *Tailoring the runtime memory model* on page 5-67

•   *Tailoring the input/output functions* on page 5-75

•   *Tailoring other C library functions* on page 5-85.

Examples of embedded applications that do not use a hosted environment are included in *Examples_directory**platform*\embedded.

Overview of semihosting dependencies

The functions shown in Table 5-2 depend directly on semihosting SWIs.

Table 5-2 Direct dependencies

Function	Description
`__user_initial_stackheap()`	*Tailoring the runtime memory model* on page 5-67. You must reimplement this function if you are using scatter-loading.
`_sys_exit()`	*Tailoring error signaling, error handling, and program exit* on page 5-51.
`_ttywrch()`	

Table 5-2 Direct dependencies (continued)

Function	Description
`_sys_command_string()`	*Tailoring the input/output functions* on page 5-75.
`_sys_close()`, `_sys_ensure()`, `_sys_iserror()`, `_sys_istty()`, `_sys_flen()`, `_sys_open()`, `_sys_read()`, `_sys_seek()`, `_sys_write()`	
`_sys_tmpnam()`	
`time()`	*Tailoring other C library functions* on page 5-85.
`remove()`	
`rename()`	
`system()`	
`clock()`, `_clock_init()`	

The functions listed in Table 5-3 depend indirectly on one or more of the functions listed in Table 5-2 on page 5-8.

Table 5-3 Indirect 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 5-51.
`__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 5-51.
`__Heap_Initialize()`	Choosing or redefining memory allocation. See *Tailoring storage management* on page 5-57.
`ferror()`, `fputc()`, `__stdout`	Retargeting the printf family. See *Tailoring the input/output functions* on page 5-75.
`__backspace()`, `fgetc()`, `__stdin`	Retargeting the scanf family. See *Tailoring the input/output functions* on page 5-75.
`fwrite()`, `fputs()`, `puts()`, `fread()`, `fgets()`, `gets()`, `ferror()`	Retargeting the stream output family. See *Tailoring the input/output functions* on page 5-75.

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.

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 5-2 on page 5-8 and Table 5-3 on page 5-9, the functions and files listed in Table 5-4 might be useful when building for a different environment.

Table 5-4 Published API definitions

File or function	Description
__main() and __rt_entry()	Initializes the runtime environment and executes the user application.
__rt_lib_init(), __rt_exit(), and __rt_lib_shutdown()	Initializes or finalizes the runtime library.
locale and CTYPE	Defines the character properties for the local alphabet. See *Tailoring locale and CTYPE* on page 5-25.
rt_sys.h	A C header file describing all the functions whose default (semihosted) implementations use the semihosting SWI.

Table 5-4 Published API definitions (continued)

File or function	Description
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 5-68 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 naming convention for libraries.

———— **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.

5.3   Building an application without the C library

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

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

- software stack checking
- low-level stdio
- signal-handling functions, signal() and raise() in signal.h
- atexit()
- alloca().

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.

5.3.1   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 not a 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.

5.3.2   Bare machine integer C

If you are writing a program in C that 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.

- 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.

5.3.3   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* on page 5-12. 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 and it occupies over 90 bytes.

5.3.4   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 5-6 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 as follows:

- 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 start to work. 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()`. (The default action if this function is not reimplemented is to create a 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 the above 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 5-15 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 5-75 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.

5.3.5   The standalone C library functions

The following sections list the include files and the functions in them 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 5-6 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 5-51 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 5-25 for more information.

math.h

Functions in this file work without the requirement for any library initialization and only require the re-implementation of `__rt_raise()`. You must call `_fp_init()` to use floating-point functions.

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 5-6 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 5-51 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()`, `perror()` and so on require high-level stdio. See *Tailoring the input/output functions* on page 5-75 for instructions.

- 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 are not available, or require implementation of a support function:

ato*()   Requires locale.

strto*()   Requires locale.

malloc()   malloc(), calloc(), realloc(), and free() require heap functions.

atexit()   Is not available.

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.

• 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.

5.4   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 an 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.

5.4.1   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, and 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.

2.   Zeroes ZI regions.

3.   Branches to __rt_entry.

If you do not want the library to do this, you can define your own __main that branches to __rt_entry as 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.

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()* for more information.

4.   Calls `exit()` with the value returned by `main()`.

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:

•   Extend the stack or heap. See *Tailoring the runtime memory model* on page 5-67.

•   Call library functions that require a callout to a user-defined function, `__rt_fp_status_addr()` or `clock()` for example. See *Tailoring other C library functions* on page 5-85.

•   Call library functions that use `LOCALE` or `CTYPE`. See *Tailoring locale and CTYPE* on page 5-25.

•   Perform floating-point calculations that require the fpu or fp library.

•   Input or output directly through low-level functions, `putc()` for example, or indirectly through high-level input/output functions and input/output support functions, `fprintf()` or `sys_open()` for example. See *Tailoring the input/output functions* on page 5-75.

•   Raise an error or other signal, `ferror` for example. See *Tailoring error signaling, error handling, and program exit* on page 5-51.

5.4.2   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 5-22
- *Tailoring error signaling, error handling, and program exit* on page 5-51.

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 retarget abort() and the stdio functions.

One solution for retargeting is to retarget 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.

5.4.3   __rt_entry

This is not a C function. The symbol __rt_entry is the starting point for a program using the ARM C library.

Implementation

__rt_entry cannot be implemented in C, because the stack has not been set up at the point this function is called. Control passes to __rt_entry after all scatter-load regions have been relocated to their execution addresses.

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 5-52.

5.4.4   __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 5-52 for details.

Returns

The function does not return.

5.4.5   __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*)

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;
};
```

5.4.6   __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.

5.5   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, for example `malloc()`, `rand()`, and `strtok()`

*   functions that do not manage a static state, but use static data in a way that is specific to their ARM implementation, 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 5-5.

Table 5-5 Callouts

Function	Description
`__rt_errno_addr()`	Called to get the address of the variable `errno`. See *__rt_errno_addr()* on page 5-53.
`__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 5-56.
The `locale` functions	The function `__user_libspace()` creates a block of private static data for the library. See *Tailoring locale and CTYPE* on page 5-25.

The functions above do not use semihosting.

See also *Tailoring the runtime memory model* on page 5-67 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. (If you are writing an operating system or a process switcher, however, you must retarget this function.)

——— **Caution** ———

The number of functions that use static data in their definitions might change in future versions of RVCT.

———————————

5.6 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.

5.6.1 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.

For implementation details, see:
- *ISO8859-1 Implementation* on page 5-26
- *Shift-JIS and UTF-8 Implementation* on page 5-26.

ISO8859-1 Implementation

To select an ISO8859-1 (Latin-1 alphabet) locale category, include a call from your application to the functions shown in Table 5-6.

Table 5-6 Default ISO8859-1 locales

Function	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 above iso8859 selections.

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

To select the locale category Shift-JIS (Japanese characters) or UTF-8 (Unicode characters), include a call from your application to the appropriate function shown in Table 5-7.

Table 5-7 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 mbtowc(), 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.

It is not possible to switch between the Shift-JIS ctype locale and other ctype locales at run time.

5.6.2 Selecting locale at run time

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 5-8.

Table 5-8 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 5-29.
_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 5-32.

Table 5-8 Locale categories (continued)

Function	Description
_get_lc_monetary()	Returns a pointer to the user-defined __lc_monetary_blk structure. See *_get_lc_monetary()* on page 5-35.
_get_lc_numeric()	Returns a pointer to the user-defined __lc_numeric_blk structure. See *_get_lc_numeric()* on page 5-37.
_get_lc_time()	Returns a pointer to the user-defined __lc_time_blk structure. See *_get_lc_time()* on page 5-38.

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.

Example 5-2 get_lc_ctype

```
void const *_get_lc_ctype(void const *null, char const *name) {
    return _findlocale(&lcctype_c_index, name);
}
```

5.6.3   Macros and utility functions

The macros and utility functions listed in Table 5-9 simplify the process of creating and using locale blocks. See the rt_locale.h file for more information.

Table 5-9 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()*.
__LC_COLLATE_DEF	Use this macro to create a block of sorting values for the character set. See *_get_lc_collate()* on page 5-32.
__LC_TIME_DEF	Use this macro to create a block of time formatting values. See *_get_lc_time()* on page 5-38.
__LC_NUMERIC_DEF	Use this macro to create a block of numeric formatting values. See *_get_lc_numeric()* on page 5-37.
__LC_MONETARY_DEF	Use this macro to create a block of monetary formatting values. See *_get_lc_monetary()* on page 5-35.
__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 5-42.

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))
```

5.6.4   _get_lc_ctype()

The ctype implementation is selected at link time to be either:
*   The C locale only. This is the default.
*   The ISO 8859 (Latin-1) locale.

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.

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.

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 */
    }
```

5.6.5 _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
};
```

5.6.6   _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, "");
}
```

5.6.7 _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.

5.6.8   _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 Time structure

```
__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`.

5.6.9   _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.

5.6.10   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 5-48.

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.

5.6.11   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 below. 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.

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 the above 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().

5.6.12 _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.

5.6.13 __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 see 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 5-29.

—— **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")
```

5.6.14   __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 see 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 5-32. See also *__LC_CTYPE_DEF* on page 5-42 for details of the side-effects of compiler optimizations.

5.6.15 __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 5-38. See also *__LC_CTYPE_DEF* on page 5-42 for details of the side-effects of compiler optimizations.

5.6.16   __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 5-37. See also *__LC_CTYPE_DEF* on page 5-42 for details of the side-effects of compiler optimizations.

5.6.17   __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 5-35. See also *__LC_CTYPE_DEF* on page 5-42 for details of the side-effects of compiler optimizations.

5.6.18 __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;
```

5.6.19 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 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 compromise 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` 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.

5.7   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 5-10. See also *Tailoring the C library to a new execution environment* on page 5-19 for additional information about application initialization and shutdown.

Table 5-10 Trap and error handling

Function	Description
`_sys_exit()`	Called, eventually, by all exits from the library. See *_sys_exit()* on page 5-52.
`errno`	Is a static variable used with error handling. See *errno* on page 5-52.
`__raise()`	Raises a signal to indicate a runtime anomaly. See *__raise()* on page 5-53.
`__rt_errno_addr()`	This function is called to obtain the address of the variable errno. See *__rt_errno_addr()* on page 5-53.
`__rt_fp_status_addr()`	This function is called to obtain the address of the fp status word. See *__rt_fp_status_addr()* on page 5-56.
`__default_signal_handler()`	Displays an error indication to the user. See *__default_signal_handler()* on page 5-55.
`_ttywrch()`	The default implementation of `_ttywrch()` is semihosted and therefore it uses the semihosting SWI. See *_ttywrch()* on page 5-55.

5.7.1   _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 reimplement 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.

5.7.2   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().

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.

5.7.3   __rt_errno_addr()

This function is called to obtain 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**)

5.7.4   __raise()

This function raises a signal to indicate a runtime anomaly.

Syntax

int `__raise`(**int** *major*, **int** *minor*)

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 5-55 for more information.

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*.

5.7.5 __rt_raise()

This function raises a signal to indicate a runtime anomaly.

Syntax

void __rt_raise(**int** *signal*, **int** *type*)

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 5-53 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*.

5.7.6   __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 5-18 on page 5-94 for those signals that are used by the libraries.

———— **Note** ————

The signals used by the libraries might change in future releases of the product.

5.7.7   _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.

5.7.8   __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.

5.8   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** ———

Users who are developing embedded systems with limited RAM might require a system that does not use the heap or any heap-using functions. Other users might require their own heap functions. There are two library functions that can be included 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 those (such as `calloc()` and `stdio`).

`__use_no_heap_region()`

> Has the same properties as `__use_no_heap()`, but in addition, 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`.

5.8.1   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 5-60.

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 5-60 for more information.

Using Heap2

The Heap2 real-time heap implementation must know how much 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 5-103).

5.8.2   Creating your own storage-management system

You can implement the heap functions in Table 5-11 to create a new
storage-management system.

Table 5-11 Heap functions

Function	Description
__Heap_Descriptor	You must define your own implementation of the abstract data type for heap. See *__Heap_Descriptor* on page 5-61.
__Heap_Initialize()	Initializes the heap. See *__Heap_Initialize()* on page 5-62.
__Heap_DescSize()	Returns the size of the __Heap_Descriptor structure. See *__Heap_DescSize()* on page 5-62.
__Heap_ProvideMemory()	Called to increase the size of the heap. See *__Heap_ProvideMemory()* on page 5-63.
__Heap_Alloc()	Allocates memory from the heap to the application. See *__Heap_Alloc()* on page 5-63.
__Heap_Free()	Returns previously allocated space to the heap. See *__Heap_Free()* on page 5-64.
__Heap_Realloc()	Adjusts the size of an already allocated block. See *__Heap_Realloc()* on page 5-64.
__Heap_Stats()	Called from __heapstats() to print statistics about the state of the heap. See *__Heap_Stats()* on page 5-65.
__Heap_Valid()	Called to perform a consistency check on the heap. See *__Heap_Valid()* on page 5-65.
__Heap_Full()	Attempts to acquire a new block from the system. You must not re-implement this function. See *__Heap_Full()* on page 5-66.
__Heap_Broken()	Called when an inconsistency in the heap is detected. See *__Heap_Broken()* on page 5-66.

5.8.3 __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()`.

5.8.4   __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.

5.8.5   __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.

ARM DUI 0232B

5.8.6   __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.

5.8.7   __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. The returned block size is typically 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.

5.8.8   __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.

5.8.9   __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 heap block. If you re-implement this function it is recommended that you maintain eight-byte alignment.

5.8.10   __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()`.

5.8.11   __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.

5.8.12   __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.

5.8.13   __Heap_Broken()

Called when an inconsistency in the heap is detected. You must not reimplement 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.

5.9   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.

5.9.1   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 while the heap grows upwards from the bottom of the region. This is the default.

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.

————— **Caution** —————

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 5-57, for how to issue a warning message if the heap or heap region is used.

———————————————

5.9.2 Controlling the runtime memory model

The behavior of the heap and stack manager can be modified by redefining the functions listed in Table 5-12.

Table 5-12 Memory model initialization

Function	Description
__user_initial_stackheap()	Returns the location of the initial heap. See *__user_initial_stackheap()* on page 5-69.
__user_heap_extend()	Returns the size and base address of a heap extra block. See *__user_heap_extend()* on page 5-70.
__user_stack_slop()	Returns the amount of extra stack. See *__user_stack_slop()* on page 5-71.

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 reimplementation. See *Tailoring static data access* on page 5-24.

5.9.3 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 5-13. 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 5-13 Memory model functions

Function	Description
__rt_stackheap_init()	Sets the application stack and initial heap. See *__rt_stackheap_init()* on page 5-71.
__rt_heap_extend()	Returns a new block of memory to add to the heap. See *__rt_heap_extend()* on page 5-73.
__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 5-74.
__rt_stack_overflow()	Handles stack overflows. (This is only required to be implemented for stack-checked variants.) See *__rt_stack_overflow()* on page 5-72.

5.9.4   __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 reimplement 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 (-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 and sl inherited from the environment 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;}
```

See also the re-implementation of this function in
Examples_Directory*platform*\Embedded\embed\retarget.c.

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.

5.9.5   __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 ATPCS 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.

5.9.6   __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.

5.9.7   __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 ATPCS. 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.

5.9.8   __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 5-73.) 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.)

5.9.9   __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 ATPCS, 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`.

5.9.10   __rt_heap_extend()

This function returns a new eight-byte aligned block of memory to add to the heap, if possible. If you reimplement the other memory model functions, you must reimplement this function. An incomplete prototype implementation is in `rt_memory.s`.

Implementation

The calling convention is ordinary ATPCS. 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 ATPCS 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 5-70). 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.

5.9.11 __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 ATPCS, 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.

5.10   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.

5.10.1   Dependencies on low-level functions

The dependencies of the higher-level function on lower-level functions is shown in Table 5-14. 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 5-14 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	-	-	-	-	-	-
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 *Reimplementing __backspace()* on page 5-77.	-	-	-	-	-	x	x	-	-	-	-

See the ISO C Reference for syntax of the low-level functions.

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 *Reimplementing __backspace()*
on page 5-77.

The standard input 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 *Examples_directory**platform*\\Embedded\\embed\\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.

Reimplementing __backspace()

The function __backspace() is used by the scanf family of functions. It must never be called directly, but reimplemented if you are retargetting 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 retruned 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.

5.10.2 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 5-15.

Table 5-15 I/O support functions

Function	Description
`_sys_open()`	*_sys_open()* on page 5-79
`_sys_close()`	*_sys_close()* on page 5-79
`_sys_read()`	*_sys_seek()* on page 5-83
`_sys_write()`	*_sys_write()* on page 5-81
`_sys_seek()`	*_sys_read()* on page 5-80
`_sys_ensure()`	*_sys_ensure()* on page 5-82
`_sys_flen()`	*_sys_flen()* on page 5-82
`_sys_istty()`	*_sys_istty()* on page 5-83
`_sys_tmpnam()`	*_sys_tmpnam()* on page 5-84
`_sys_command_string()`	*_sys_command_string()* on page 5-84

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.

5.10.3 _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.

5.10.4 _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.

5.10.5 _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.

5.10.6 _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 reimplement 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.

5.10.7   **_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.

5.10.8   **_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.

5.10.9 _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.

5.10.10 _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 not an interactive device

1   There is an interactive device

other   An error occurred.

5.10.11 _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*.

5.10.12 _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.

5.11 Tailoring other C library functions

Implementation of the following ISO standard functions depends entirely on the target operating system. None of the functions listed below is used internally by the library. So if any of these functions are not implemented, only those applications fail that call the function directly.

The target-dependent ISO C library functions are listed in Table 5-16.

Table 5-16 ISO C library functions

Function	Description
clock() and _clock_init()	*clock()* on page 5-86 and *_clock_init()* on page 5-86
time()	*time()* on page 5-87
remove()	*remove()* on page 5-87
rename()	*rename()* on page 5-88
system()	*system()* on page 5-88
getenv()	*getenv()* on page 5-89
__getenv_init()	*_getenv_init()* on page 5-89

The default implementation of these functions is semihosted. That is, each function uses the semihosting SWI.

clock() and _clock_init() must be reimplemented together or not at all.

5.11.1   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.

5.11.2   _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.

                   ARM DUI 0232B

5.11.3 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*`.

5.11.4 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.

5.11.5   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.

5.11.6   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.

5.11.7   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().

5.11.8   _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.

5.12   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, two divide routines are provided with the ARM library.

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** ──────

Because it uses the CLZ instruction and the DSP multiplies, the real-time division routine only works on ARMv5TE and above. If you reference `__use_realtime_division` and compile and link for a core that does not support these, the linker displays an error similar to:

```
Error: L6218E: Undefined symbol
__realtime_division_only_works_on_architecture_5E__
(referred from myobj.o)
```

─────────────────────────────

5.13   ISO implementation definition

This section describes how the libraries fulfill the requirements of the ISO specification.

5.13.1   ISO C library implementation definition

The ISO C library variants are listed in *Library naming conventions* on page 5-106.

The ISO specification leaves some details to the implementors, but requires their implementation choices to be documented. The implementation details are described in this section.

*   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_*

The following functions test for character values in the range EOF (–1) to 255 (inclusive):

*   `isalnum()`
*   `isalpha()`
*   `iscntrl()`
*   `islower()`
*   `isprint()`
*   `isupper()`
*   `ispunct()`.

Mathematical functions

The mathematical functions shown in Table 5-17, when supplied with out-of-range arguments, respond in the way shown.

Table 5-17 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

Table 5-17 Mathematical functions (continued)

Function	Condition	Returned value	Error number
pow(x,y)	x=1, y=Inf	QNaN	EDOM
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 5-17 on page 5-92, all functions return QNaN when passed QNaN and throw an invalid operation exception when passed SNaN.

Signal function

The signals listed in Table 5-18 are supported by the signal() function.

Table 5-18 Signal function signals

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 5-18 Signal function signals (continued)

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 5-51, Chapter 6 *Floating-point Support*, and Chapter 6 *Floating-point Support* for more details on floating-point.

For all the signals in Table 5-18 on page 5-94, when a signal occurs, if the handler points to a function, the equivalent of signal(sig, SIG_DFL) is executed before the call to 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 in Table 5-19.

Table 5-19 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, required to be specified in an ISO-compliant implementation, 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().

5.13.2 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 5-21 on page 5-99. 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 the *install_directory*\Documentation\RogueWave\1.0\release.

The standard C++ library is distributed in binary form only.

The requirements that the C++ library places on the C library are described in Table 5-20.

Table 5-20 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()

The most important features missing from this release are described in Table 5-21.

Table 5-21 Standard C++ library differences

Standard	Implementation differences
Unimplemented features	Support functions for unimplemented language features, `class bad_cast` for example, are unlikely to be functional.
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.
Complex default template arguments	Not supported. Complex default template argument definitions are where a type parameter has a default instantiation involving an earlier type parameter. When you request a template that the standard says is defined with a complex default (such as instantiating class queue), you must always supply a value for each template parameter. No defaults are present.
Exceptions	Not supported.
typeinfo	Limited support. typeinfo is supported in a basic way by the ARM C++ library additions.

5.14   C library extensions

This section describes the ARM-specific library extensions and functions defined by the C99 draft standard (*ISO/IEC 9899:1999E*). The extensions are summarized in Table 5-22. The headers <stdint.h> and <inttypes.h> from C99 are also available.

Table 5-22 Extensions

Function	Header file definition	Extension
atoll()	stdlib.h	C99 standard
strtoll() on page 5-101	stdlib.h	C99 standard
strtoull() on page 5-101	stdlib.h	C99 standard
snprintf() on page 5-101	stdio.h	C99 standard
vsnprintf() on page 5-102	stdio.h	C99 standard
lldiv() on page 5-102	stdlib.h	C99 standard
llabs() on page 5-102	stdlib.h	C99 standard
alloca() on page 5-103	alloca.h	C99 and others
_fisatty() on page 5-103	stdio.h	ARM-specific
__heapstats() on page 5-103	stdlib.h	ARM-specific
__heapvalid() on page 5-105	stdlib.h	ARM-specific

5.14.1   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*)

5.14.2   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)

5.14.3   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)

5.14.4   snprintf()

snprintf() 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, ...)

                5-101

5.14.5 vsnprintf()

vsnprintf() 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)
```

5.14.6 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)
```

5.14.7 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)
```

5.14.8 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*)

5.14.9 _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 5-75) on the underlying file handle.

This function is an ARM-specific library extension.

Syntax

int _fisatty(FILE ∗*stream*)

The return value indicates the stream destination:

0   A file.
1   A terminal.
Negative   An error.

5.14.10 __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 5-65). The ARM default implementation 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 heapstats output

```
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. You must therefore ensure you have already sent some output to stderr in the above 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 ARM-specific library extension.

Syntax

```
void __heapstats(int (*dprint)( void*param,
                 char const *format,...), void* param)
```

5.14.11 __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 will fail if you have not already written to the stream.

This function is an ARM-specific library extension.

Syntax

```
int __heapvalid(int (*dprint)( void*param, char const *format,...), void* param,
                int verbose)
```

5.15   Library naming conventions

The filename identifies how the variant was built as follows:

root_<arch><fpu><stack><entrant>.<endian>

The values for the fields of the name and the relevant build options are listed below:

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.
	fz	Behaves like the fj library, but additionally flushes denormals and infinities to zero. This library behaves like the ARM VFP in Fast mode. This is the default.
	g	IEEE complaint library with configurable rounding mode and all IEEE exceptions.
	m	Transcendental math functions.
	cpp	High-level C++ functions that do not require fp arithmetic.
	cpprt	The C++ runtime libraries.
arch	a	An ARM library.
	t	A Thumb library (`--apcs /interwork`). See *Thumb C libraries* on page 5-108.
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`).
	_	Not applicable.
endian	l	Little-endian (`--li`).
	b	Big-endian (`--bi`).

The C library names are c_{a,t}__{s,u}{e,n}

c_a__se   ARM, stack checking, reentrant

c_a__sn   ARM, stack checking, not reentrant

c_a__ue   ARM, no stack checking, reentrant

c_a__un   ARM, no stack checking, not reentrant (base PCS)

c_t__se   Thumb, stack checking, reentrant

c_t__sn   Thumb, stack checking, not reentrant

c_t__ue   Thumb, no stack checking, reentrant

c_t__un   Thumb, no stack checking, not reentrant (base PCS).

The standard FPLIB names are f_{a,t}[fm, vp, _m, _p]

f_a_p   ARM, soft VFP, pure-endian

f_a   ARM, used with --fpu none

f_t_p   Thumb, soft VFP, pure-endian double

f_t   Thumb, used with --fpu none.

The --fpmode ieee_no_fenv names are fj_{a,t}{fm,vp,_m,_p}

fj_a_p   ARM, soft VFP, pure-endian

fj_t_p   Thumb, soft VFP, pure-endian double.

The --fpmode std and --fpmode fast names are fz_{a,t}{fm,vp,_m,_p}

fz_a_p   ARM, soft VFP, pure-endian

fz_a   ARM, used with --fpu none

fz_t_p   Thumb, soft VFP, pure-endian double

fz_t   Thumb, used with --fpu none.

The standard IEEE names are g_{a,t}{fm, vp, _m, _p}

g_a_p   ARM, soft VFP

g_t_p   Thumb, pure-endian double.

The MATHLIB names are m_{a,t}{fm, vp, _m, _p}{s,u}

m_a_ps   ARM, soft VFP, pure-endian, stack checking

m_a_pu   ARM, soft VFP, pure-endian, no stack checking

m_t_ps   Thumb, soft VFP, pure-endian, stack checking

m_t_pu   Thumb, soft VFP, pure-endian, no stack checking.

5.15.1   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 or interworking, or for ARMv5T or later.

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.

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 6-8
- *The math library, mathlib* on page 6-24
- *IEEE 754 arithmetic* on page 6-30.

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 6-30 for details of the ARM implementation of the standard.

The compiler implements the floating-point calculations in software. For example, the compiler option `--fpu softvfp` specifies that coprocessor instructions are to be implemented in software.

 ARM DUI 0232B

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 have to provide a set of procedure calls to do floating-point arithmetic. These procedures are in the software floating-point library, fplib.

Floating-point routines have names like _dadd (add two **double**s) and _fdiv (divide two **float**s). The complete list is given in:

- Table 6-1 on page 6-4
- Table 6-2 on page 6-5
- Table 6-3 on page 6-6
- Table 6-4 on page 6-7.

User programs can call these routines directly. Even in environments with a coprocessor, the routines are provided, though they are typically only a few instructions long (as 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 are declared in the header file rt_fp.h. You can include this file if you want to call an fplib routine directly.

A complete list of the fplib routines is provided on the following pages.

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
```

6.2.1 Arithmetic on numbers in a particular format

The routines in Table 6-1 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
_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 above 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 6-5

6.2.2   Conversions between floats, doubles, and ints

The routines in Table 6-2 perform conversions between number formats, excluding **long long**s.

Table 6-2 Number format conversion routines

Function	Argument type	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.

6.2.3   Conversions between long longs and other number formats

The routines in Table 6-3 perform conversions between **long long**s and other number formats.

Table 6-3 Conversion routines involving long long format

Function	Argument type	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.

6.2.4   Floating-point comparisons

The routines in Table 6-4 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*).

6.3 Controlling the floating-point environment

This section describes the functions you can use to control the ARM floating-point environment. With these functions, you can change the rounding mode, enable and disable trapping of exceptions, and install your own custom exception trap handlers.

ARM supplies several different interfaces to the floating-point environment, for compatibility and porting ease.

6.3.1 The __ieee_status function

ARM supports a second interface to the status word, similar to the __fp_status function, but the second interface sees the same status word in a different layout. This call is called __ieee_status, and it is generally the most efficient function to use for modifying the status word for VFP. (__fp_status is more efficient on FPA systems.) __ieee_status is defined in fenv.h.

Like __fp_status, __ieee_status has the prototype:

```
unsigned int __ieee_status(unsigned int mask,
                           unsigned int flags);
```

However, the layout of the status word as seen by __ieee_status is different from that seen by __fp_status (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		
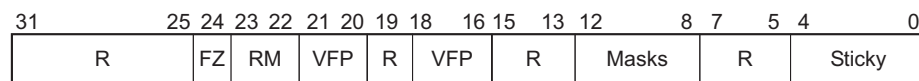
Figure 6-1 IEEE status word layout

The fields in Figure 6-1 are as follows:

- Bits 0 to 4 are the sticky flags, exactly as described in *The __fp_status function* on page 6-10.

- Bits 8 to 12 are the exception mask bits, exactly as described in *The __fp_status function* on page 6-10, but in a different place.

- Bits 16 to 18, and bits 20 and 21, are used by VFP hardware 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 6-5).

Table 6-5 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 5-106.

• 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.

• Bits marked R are reserved.

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   The __fp_status function

Previous versions of the ARM libraries implemented a function called `__fp_status`, that manipulated a status word in the floating-point environment. ARM still supports this function, for backwards compatibility. It is defined in `stdlib.h`.

`__fp_status` has the following prototype:

unsigned int `__fp_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 (Table 6-6).

Table 6-6 Status word bit modification

Bit of mask	Bit of flags	Effect
0	0	Leave alone
0	1	Toggle
1	0	Set to 0
1	1	Set to 1

The layout of the status word as seen by `__fp_status` is shown in Figure 6-2 on page 6-11.

31		24	23	21	20		16	15	13	12		8	7		5	4		0
System ID			R		Masks			R		FPA only			R			Sticky		
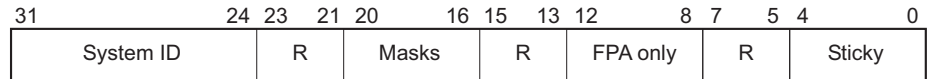
Figure 6-2 Floating-point status word layout

The fields in Figure 6-2 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`) control various aspects of the FPA floating-point coprocessor. Any attempt to write to these bits is ignored if there is no FPA in your system.

- Bits 16 to 20 (values `0x10000` to `0x100000`) 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 6-35.

- Bits 24 to 31 contain the system ID that cannot be changed. It is set to `0x40` for software floating-point, to `0x80` or above for hardware floating-point, and to 0 or 1 if a hardware floating-point environment is being faked by an emulator.

- 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 three 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`.

The _controlfp function

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 `__fp_status` and `__ieee_status`, where you can toggle a bit by setting a zero in the mask word and a one in the flags word.

The macros you can use to form the arguments to `_controlfp` are given in Table 6-7.

Table 6-7 _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
`_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);`

The _clearfp function

The function _clearfp clears all five exception sticky flags, and returns their previous value. The macros given in Table 6-7 on page 6-13, 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. The macros given in Table 6-7 on page 6-13, for example _EM_INVALID, _EM_ZERODIVIDE, can be used to test bits of the returned result.

_statusfp has the following prototype:

unsigned _statusfp(**void**);

6.3.4 C9X-compatible functions

In addition to the above functions, ARM also supports a set of functions defined in the C9X draft standard. These 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. ARM defines them to be structure types, for details see *ARM extensions to the C9X interface* on page 6-17.

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 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, as one of the macros defined above. 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 extensions to the C9X interface

ARM 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. ARM 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 the above structure are:

- `statusword` is the same status variable that the function `__ieee_status` sees, laid out in the same format (see *The __ieee_status function* on page 6-8).

- 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 *The __ieee_status function* on page 6-8).

- 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 6-8.

Table 6-8 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**
`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 6-9.

Table 6-9 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 6-10.

Table 6-10 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.
FE_EX_FN_CMP	Compare.
FE_EX_FN_CVT	Convert between formats.
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 6-11.

Input and output types are the same for all operations except Compare and Convert.

Table 6-11 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 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.

A handler function that does this is shown in Example 6-1.

Example 6-1

```
__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);
}
```

Install the handler function as follows:

```
fenv_t env;
fegetenv(&env);
env.statusword |= FE_IEEE_MASK_INVALID;
env.invalid_handler = myhandler;
fesetenv(&env);
```

After the handler is installed, dividing 0.0 by 0.0 returns 1.0.

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 6-22, 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

Trigonometric functions in mathlib use range reduction to bring large arguments within the range 0 to 2π. ARM 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.

In addition to the functions defined by the ISO C standard, mathlib provides the following functions:

- *Inverse hyperbolic functions (acosh, asinh, atanh)* on page 6-25
- *Cube root (cbrt)* on page 6-25
- *Copy sign (copysign)* on page 6-25
- *Error functions (erf, erfc)* on page 6-25
- *One less than exp(x) (expm1)* on page 6-26
- *Determine if a number is finite (finite)* on page 6-26
- *Gamma function (gamma, gamma_r)* on page 6-26
- *Hypotenuse function (hypot)* on page 6-26
- *Return the exponent of a number (ilogb)* on page 6-27
- *Determine if a number is a NaN (isnan)* on page 6-27
- *Bessel functions of the first kind (j0, j1, jn)* on page 6-27
- *The logarithm of the gamma function (lgamma, lgamma_r)* on page 6-27
- *Logarithm of one more than x (log1p)* on page 6-28
- *Return the exponent of a number (logb)* on page 6-28
- *Return the next representable number (nextafter)* on page 6-28
- *IEEE 754 remainder function (remainder)* on page 6-28
- *IEEE round-to-integer operation (rint)* on page 6-28
- *Scale a number by a power of two (scalb, scalbn)* on page 6-29
- *Return the fraction part of a number (significand)* on page 6-29
- *Bessel functions of the second kind (y0, y1, yn)* on page 6-29.

6.4.1   Inverse hyperbolic functions (acosh, asinh, atanh)

```
double acosh(double x);
double asinh(double x);
double atanh(double x);
```

These functions are the inverses of the ISO-required cosh, sinh and tanh:

- Because cosh is a symmetric function (that is, it returns the same value when applied to *x* or *–x*), acosh always has a choice of two return values, one positive and one negative. 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.

6.4.2   Cube root (cbrt)

```
double cbrt(double x);
```

This function returns the cube root of its argument.

6.4.3   Copy sign (copysign)

```
double copysign(double x, double y);
```

This function replaces the sign bit of *x* with the sign bit of *y*, and returns the result. It causes no errors or exceptions, even when applied to NaNs and infinities.

6.4.4   Error functions (erf, erfc)

```
double erf(double x);
double erfc(double x);
```

These functions compute the standard statistical error function, related to the Normal distribution:

- 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.

6.4.5 One less than exp(*x*) (expm1)

```
double expm1(double x);
```

This function computes *e* to the power *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.

6.4.6 Determine if a number is finite (finite)

```
int finite(double x);
```

This function returns 1 if *x* is finite, and 0 if *x* is infinite or NaN. It does not cause any errors or exceptions.

6.4.7 Gamma function (gamma, gamma_r)

```
double gamma(double x);
double gamma_r(double x, int *);
```

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 6-27).

———— **Note** ————

Despite their names, these functions compute the logarithm of the gamma function, not the gamma function itself.

6.4.8 Hypotenuse function (hypot)

```
double hypot(double x, double y);
```

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* could cause $x * x + y * y$ to overflow even though its square root would not.

`hypot` returns an `ERANGE` error when the result does not fit in a **double**.

6.4.9   Return the exponent of a number (ilogb)

int ilogb(**double** x);

This function returns the exponent of *x*, without any bias, so ilogb(1.0) would return 0, and ilogb(2.0) would return 1, and so on.

When applied to 0, ilogb returns -0x7FFFFFFF. When applied to a NaN or an infinity, ilogb returns +0x7FFFFFFF. ilogb causes no exceptions or errors.

6.4.10   Determine if a number is a NaN (isnan)

int isnan(**double** x);

This function returns 1 if *x* is *Not a Number* (NaN), and 0 otherwise. It causes no exceptions or errors.

6.4.11   Bessel functions of the first kind (j0, j1, jn)

double j0(**double** x);
double j1(**double** x);
double jn(**int** n, **double** x);

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.*

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.

6.4.12   The logarithm of the gamma function (lgamma, lgamma_r)

double lgamma(**double** x);
double lgamma_r(**double** x, **int** *sign);

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*.

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.

6.4.13   Logarithm of one more than x (log1p)

double log1p(**double** x);

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.

6.4.14   Return the exponent of a number (logb)

double logb(**double** x);

This function is similar to ilogb, but returns its result as a **double**. It can therefore return special results in special cases.

- logb(NaN) is a quiet NaN.
- logb(infinity) is +infinity.
- logb(0) is –infinity, and causes a Divide by Zero exception.

logb is the same function as the Logb function described in the IEEE 754 Appendix.

6.4.15   Return the next representable number (nextafter)

double nextafter(**double** x, **double** y);

This function returns the next representable number after x, in the direction toward y. If x and y are equal, x is returned.

6.4.16   IEEE 754 remainder function (remainder)

double remainder(**double** x, **double** y);

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).

6.4.17   IEEE round-to-integer operation (rint)

double rint(**double** x);

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).

6.4.18   Scale a number by a power of two (scalb, scalbn)

```
double scalb(double x, double n);
double scalbn(double x, int n);
```

These functions return *x* times two to the power *n*. The difference between the functions is whether *n* is passed in as an **int** or as a **double**.

scalb is the same function as the scalb function described in the IEEE 754 Appendix. Its behavior when *n* is not an integer is undefined.

6.4.19   Return the fraction part of a number (significand)

```
double significand(double x);
```

This function returns the fraction part of *x*, as a number between 1.0 and 2.0 (not including 2.0).

6.4.20   Bessel functions of the second kind (y0, y1, yn)

```
double y0(double x);
double y1(double x);
double yn(int, double);
```

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*.

If *x* is positive and exceeds π times 2^{52}, these functions return an ERANGE error, denoting total loss of significance in the result.

6.5   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 ARM.

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.

31	30 23	22 0
S	Exp	Frac

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 the above 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		
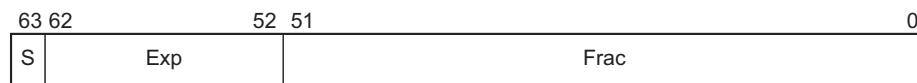
Figure 6-4 IEEE 754 double-precision floating-point format

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 6-12 and Table 6-13 on page 6-33.

Table 6-12 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	Signalling 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 decimal places 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, signalling NaNs and quiet NaNs. Quiet NaNs have a 1 in the first bit of Frac, and signalling NaNs have a zero there. The difference is that signalling NaNs cause an exception (see *Exceptions* on page 6-35) when used, whereas quiet NaNs do not.

Table 6-13 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	Signalling NaN	f
0x7FF80000 00000000	0	0x7FF	100...000	Quiet NaN	f

a. to f. For footnotes, see Table 6-12 on page 6-32.

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. (Hardware floating-point environments and the enhanced floating-point libraries, g_avp for example, support all four rounding modes. See *Library naming conventions* on page 5-106.)

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 signalling 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 the above 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 6-12 on page 6-32) 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 depends 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 ARM choice of implementation options 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, and hardware floating-point systems, 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. Semihosting enables code running on an ARM® target to use the I/O facilities on a host computer that is running RealView Debugger. Examples of these facilities include the keyboard input, screen output, and disk I/O. This chapter 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 7-21.

7.1   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.

Semihosting is implemented by a set of defined 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.

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 shows an overview of semihosting.
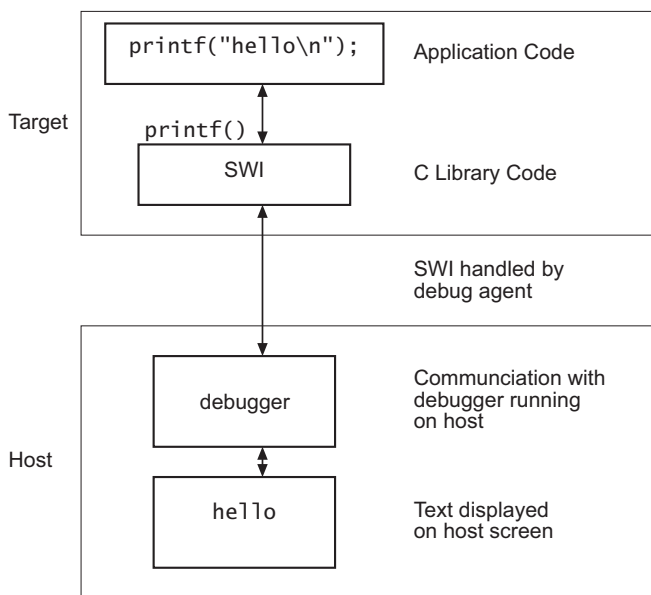
Figure 7-1 Semihosting overview

The semihosting SWI interface is common across all debug agents provided by ARM Limited. Semihosted operations work under RVI-ME without any requirement for porting.

For further information on semihosting and the C libraries, see Chapter 5 *The C and C++ Libraries.*.

7.1.1 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 Limited.
0x32-0xFF	These are reserved for future use by ARM Limited.
0x100-0x1FF	These are reserved for user applications. They are not used by ARM Limited.
	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. They are not currently used and not recommended for use.

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 it is best to use the operation names that are defined in semihost.h. You can define the operation names with an EQU directive. For example:

```
SYS_OPEN    EQU 0x01
SYS_CLOSE   EQU 0x02
```

 ARM DUI 0232B

7.2 Semihosting implementation

The functionality provided by semihosting is basically the same on all debug hosts. The implementation of semihosting, however, differs between hosts.

7.2.1 RVI-ME

When using RVI-ME , semihosting is handled with either a real SWI exception handler, or by emulating a handler using breakpoints. See your Getting Started guide, for more details on semihosting with RVI-ME.

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 ATPCS 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 7-9	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 7-10	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 7-11	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 7-12	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 7-17	Get the value of the C library errno variable
SYS_GET_CMDLINE (0x15) on page 7-18	Get the command-line used to call the executable

7.3.1 SYS_OPEN (0x01)

Open a file on the host system. The file path is specified either as relative to the current directory of the host process, or absolutely, using the path conventions of the host operating system.

The 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 RealView 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 RealView 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 end-of-file 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 pathname 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 file name.

word 3   This is a pointer to the new file name.

word 4   This is the length of the new file name.

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 7-20 and *SYS_TICKFREQ (0x31)* on page 7-20).

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 the 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.

           ARM DUI 0232B

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 Chapter 5 *The C and C++ Libraries* 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 Structure of data block and return values

```
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.

                                   ARM DUI 0232B

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 7-22.

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 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. 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

Table 7-4 Software reason codes (continued)

Name (#defined in adp.h)	Hexadecimal value
ADP_Stopped_InternalError	*0x20024
ADP_Stopped_UserInterruption	0x20025
ADP_Stopped_ApplicationExit	0x20026
ADP_Stopped_StackOverflow	*0x20027
ADP_Stopped_DivisionByZero	*0x20028
ADP_Stopped_OSSpecific	*0x20029

* next to values in Table 7-4 on page 7-22 indicates that the value is not supported by RealView Debugger. RealView Debugger reports an Unhandled ADP_Stopped exception for these values.

Return

No return is expected from these calls. However, it is possible for RealView Debugger to request that the application continue by performing an RDI_Execute request or equivalent. In this case, execution continues with the registers as they were on entry to the SWI, or as subsequently modified by RealView Debugger.

ARM DUI 0232B

Appendix A
Via File Syntax

This appendix describes the syntax of via files accepted by the ARM® development tools, such as the ARM compiler, linker, assembler, and `fromELF` 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`
- the assembler, `armasm`
- the linker, `armlink`
- the ARM librarian, `armar`.

You can specify a via file from the command line using the `-via` tool 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 white space, 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 white space. 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 pathnames that contain spaces. For example:

  ```
  --libpath c:\Program Files\ARM\RVCT\Data\version\release\lib   (three words)
  --libpath "c:\Program Files\ARM\RVCT\Data\version\release\lib" (two words)
  ```

  Apostrophes can be used to delimit words that contain quotes. For example:

  ```
  -DNAME='"RealView"' (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:\Program Files\ARM\RVCT\Data\version\release\lib"
  ```

  is treated as the single word:

  ```
  -IC:\Program Files\ARM\RVCT\Data\version\release\lib
  ```

- Lines beginning with a semicolon (;) or a hash (#) character as the first nonwhitespace character are comment lines. If a semicolon or hash character appears anywhere else in 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 -D*symbol*="*value*" must be delimited with a single quote, either as '-D*symbol*="*value*"' or as -D*symbol*='"*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 sections:

- *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 section provides references where applicable.

————————————

The following subsections correspond to the relevant sections of subclause G3. They describe aspects of the ARM® C compiler and ISO 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-5
- *Structures, unions, enumerations, and bitfields* on page B-6
- *Qualifiers* on page B-6
- *Declarators* on page B-7
- *Statements* on page B-7
- *Preprocessing directives* on page B-7
- *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 2-45) 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++.
Warning	This indicates unusual conditions in your code that might indicate a problem.
Error	This indicates violations in the syntactic or semantic rules of the C or C++ language.
Serious error	This is a catastrophic error and indicates a problem that causes the compilation to stop.

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
- *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 white space, except where the white space 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 unhosted 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).

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 stderrto *filename*.

>> *filename*   This appends stdout to *filename*.

>>& *filename* This appends both stdout and stderr to *filename*.

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 3-32 for details.

B.1.4 Characters

See *Character sets and identifiers* on page 3-32 for details.

B.1.5 Integers

See *Integer* on page 3-35 for details.

B.1.6 Floating-point

See *Float* on page 3-35 for details.

B.1.7 Arrays and pointers

See *Arrays and pointers* on page 3-35 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 3-11 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 3-32.

Unions

See *Unions* on page 3-37 for details.

Enumerations

See *Enumerations* on page 3-38 for details.

Padding and alignment of structures

See *Structures* on page 3-38 for details.

Bitfields

See *Bitfields* on page 3-40 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 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 *About the runtime libraries* on page 5-2.

Appendix C
Standard C++ Implementation Definition

The majority of the language features described in the ISO/IEC standard for C++ are supported by the ARM® compiler 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++.

───── **Note** ─────

This section 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 3-19.

This appendix 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.

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

If a pure virtual function is called, the signal **SIGPVFN** is raised. The default signal handler prints an error message and exits. See *__default_signal_handler()* on page 5-55.

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	1-13	Yes
Templates	14	Yes, with the exception of export templates
Exceptions	15	No
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 slight 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 5-97.

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 list 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. A summary is given in Table D-1. A limit of memory indicates that no limit is imposed by the ARM compiler, 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	1024
Number of initial characters in an external identifier.	1024	1024
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	268435454
Nesting levels of #include file.	256	memory

Table D-1 Implementation limits (continued)

Description	Recommended	ARM
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. (A file with more lines gives wrapped line numbers in messages because the internal format for line numbers is a 16-bit unsigned short.)	65536
Maximum number of relocatable references in a single translation unit.	memory
Maximum number of virtual registers.	65536
Maximum number of overload arguments.	256
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 third column of the table gives the numerical value of the range endpoint. The fourth 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
LONG_LONG_MAX	Maximum value of **long long**	9.2E+18	0x7FFFFFFF FFFFFFFF
LONG_LONG_MIN	Minimum value of **long long**	−9.2E+18	0x80000000 00000000
ULONG_LONG_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.

——— **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.

Table D-4 Floating-point limits

Constant	Meaning	Value
FLT_MAX	Maximum value of **float**	3.40282347e+38F
FLT_MIN	Minimum value of **float**	1.17549435e–38F
DBL_MAX	Maximum value of **double**	1.79769313486231571e+308
DBL_MIN	Minimum value of **double**	2.22507385850720138e–308
LDBL_MAX	Maximum value of **long double**	1.79769313486231571e+308
LDBL_MIN	Minimum 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

Table D-4 Floating-point limits (continued)

Constant	Meaning	Value
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

Appendix E
Older Compiler Options

This appendix describes those options from the legacy ARM® compilers that are provided in the RealView® Compilation Tools (RVCT) v2.0 compiler for backwards compatibility. It is intended that you migrate your use of these options to the equivalent options in the RVCT v2.0 compiler.

It contains the following section:

- *Using the older compiler options* on page E-2
- *Command Syntax Supported with --old_cfe* on page E-3
- *Mapping old compiler options to the new options* on page E-12.

E.1   Using the older compiler options

When you compile your C code using the `--old_cfe` option, only the options supported in the legacy compilers have an effect. You can, however, use some of the newly supported C extensions, such as:

* `pragma arm` and `pragma thumb` pragmas
* `--arm` and `--thumb` options.

———— **Note** ————

You can compile only C code when you specify the `--old_cfe` option.

For details of the older C compiler options, see *Command Syntax Supported with --old_cfe* on page E-3.

E.2 Command Syntax Supported with --old_cfe

This section describes the options in the legacy ARM compilers that are still supported. It includes:

- *Setting the source language*
- *Specifying output format*
- *Warning message options* on page E-4
- *Specifying additional checks* on page E-9
- *Error message options* on page E-10.

E.2.1 Setting the source language

The following options are used for setting the source language:

-ansi
-ansic   Enables compilation of C rather than C++, specifically the C90 ISO version of C. The default mode is a fairly strict ISO compiler, but without some of the inconvenient features of the ISO standard. There are also some minor extensions supported, for example // in comments and $ in identifiers.

E.2.2 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 source files. They are available when you specify the --old_cfe option.

-list   Creates a listing file consisting of lines of source interleaved with error and warning messages. The deprecated options -fi, -fj, and -fu can be used to control the contents of this file.

> ——— **Note** ———
>
> The -list option does not accept a pathname for the output file. You must rename previous versions of list files if you do not want to overwrite them.

-fi   Deprecated. Used with -list to list the lines from any files included with directives of the form #include "file".

-fj   Deprecated. Used with -list to list the lines from any files included with directives of the form #include <file>.

-fu   Deprecated. Used with -list to list source that was not preprocessed.

By default, if you specify -list, the compiler lists the source text as seen by the compiler after preprocessing. If you specify -fu, the unexpanded source text is listed. For example:

```
p = NULL;        /* assume #defined NULL 0 */
```

If -fu is not specified, this is listed as:

```
p = 0;
```

If -fu is specified, it is listed as:

```
p = NULL;
```

E.2.3   Warning message options

The compiler issues warnings about potential portability problems and other hazards. The compiler options enable you to turn off specific warnings. 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, it is better to check the code than to switch off warnings.

The options are on by default, unless specified otherwise.

See also *Specifying additional checks* on page E-9 for descriptions of additional warning messages.

The general form of the -W compiler option is:

-W[*options*][+][*options*]

where the *options* field contains zero or more characters.

If the + character is included in the characters following the -W, the warnings corresponding to any following letters are enabled rather than suppressed.

You can specify several options at the same time. For example:

-Wad+fg

turns off the warning messages specified by a and d, and turns on the warning messages specified by f and g.

The following warning message options are available. Any message listed for an option is the old style message. To find the equivalent message displayed when you do not specify the --old_cfe option, see *Mapping of compiler messages* on page E-13.

-W   Suppresses all warnings. If one or more letters follow the option, only the warnings controlled by those letters are suppressed.

-Wa   Suppresses the warning:

C2916W: Use of '=' in a condition context

This warning is normally given when the compiler finds a statement such as:

if (a = b) {...

where it is possible that one of the following is intended:

if ((a = b) != 0) {...
if (a == b) {...

-Wb   Suppresses the warning messages that are issued for extensions to the ISO standard, for example:

C2485W: ANSI C forbids bit field type

Examples of errors include:

- using an unwidened type in an ISO C assignment

- specifying bitfields with a type of **char**, **short**, **long**, or **long long**

- specifying **char**, **short**, **float**, or **enum** arguments to variadic functions such as va_start().

-Wd   Suppresses the warning message:

C2215W: Deprecated declaration foo() - give arg types

This warning is normally given when a declaration without argument types is encountered in ISO C mode.

In ISO C, declarations like this are deprecated. However, it is sometimes useful to suppress this warning when porting old code.

In C++, void foo(); means void foo(void); and no warning is generated.

-We   Suppresses the warning messages given when an extended constant expression is used in an initializer (see *C language extensions* on page 3-19) that other C compilers are not required by the standard to accept:

C2564W: extended constant initialiser used

-Wf   Suppresses the message:

C2207W: Inventing extern int foo()

This is an error in C++ and cannot be suppressed. It is a warning in ISO C, and suppressing this message can be useful when compiling old-style C in ISO C mode.

-Wg   Suppresses the warning given when an unguarded header file is #included.

C2819W: Header file not guarded against multiple inclusion

This warning is off by default. It can be enabled with -W+g. An unguarded header file is a header file not wrapped in a declaration such as:

```
#ifndef foo_h
#define foo_h
/* body of include file */
#endif
```

-Wi   Deprecated. Suppresses the implicit constructor warning (C++ only).

```
C2887W: implicit constructor 'struct X'()
```

It is issued when the code requires a constructor to be invoked implicitly, for example:

```
struct X { X(int); };
X x = 10;        // actually means, X x = X(10);
                 // See the Annotated C++
                 // Reference Manual p.272
```

This warning is switched off by default. It can be enabled with -W+i.

―――― **Note** ――――

This option is not available when compiling with the --old_cfe option.

-Wk   Turns off the warning:

```
C2621W: double constant automatically converted to float
```

These warnings are given when the default type of unqualified floating-point constants is changed by the -auto_float_constants option. This warning is switched on by default.

-Wl   Deprecated. Turns off the warning:

```
C2915W: lower precision in wider context
```

when code similar to the following is found:

```
long x; int y, z; x = y*z
```

where the multiplication yields an **int** result that is then widened to **long**. This warning indicates a potential problem when either the destination is **long long** or where the code has been ported to a system that uses 16-bit integers or 64-bit longs. This option is off by default. It can be enabled with -W+l.

-Wm   Deprecated. Suppresses warnings about multiple-character **char** constants:

```
C2203W: non-portable - not 1 char in '...'
```

-Wn   Deprecated. Suppresses the warning message:

            ARM DUI 0232B

C2921W: implicit narrowing cast

This warning is issued when the compiler detects the implicit narrowing of a long expression in an **int** or **char** context, or detects the implicit narrowing of a floating-point expression in an integer or narrower floating-point context.

Such implicit narrowing casts are almost always a source of problems when moving code that has been developed on a 32-bit system to a system where **int** occupies 16 bits and **long** occupies 32 bits. This option is off by default.

-Wo   Suppresses warnings for implicit conversion to signed **long long** constants:

C2201W: '*Number*' treated as '*Number*ll'

-Wp   Deprecated, Suppresses the warning message:

C2812W: Non-ANSI #include <…>

The ISO C standard requires that you use #include <…> for ISO C headers only. However, it is useful to disable this warning when compiling code not conforming to this aspect of the standard. This warning is suppressed by default unless you specify the -strict option.

-Wq   Suppresses warnings in C++ constructor initialization order:

C2252W: base *class* will be initialized in declaration order instead of the order written

C2253W: member *name* will be initialized in declaration order instead of the order written

———— **Note** ————

This option is not available when compiling with the --old_cfe option.

-Wr   Suppresses the implicit virtual warning (C++ only) issued when a non-virtual member function of a derived class hides a virtual member of a parent class. For example:

```
struct Base { virtual void f(); };
struct Derived : Base { void f(); };
```

generates the following warning:

C2997W: 'Derived::f()' inherits implicit virtual from 'Base::f()'

Adding the **virtual** keyword in the derived class prevents the warning.

--- **Note** ---

This option is not available when compiling with the `--old_cfe` option.

-Ws   Suppresses warnings generated when the compiler inserts padding in a **struct**, for example:

`C2221W: padding inserted in struct 's'`

This warning is off by default. It can be enabled with -W+s.

-Wt   Deprecated. Suppresses the unused **this** warning. This warning is issued when the implicit **this** argument is not used in a non-static member function. It is applicable to C++ only. The warning can also be avoided by making the member function a static member function. The default is off. For example:

```
struct T {
        int f() { return 42; }
};
```

results in the following warning:

`C2924W: 'this' unused in non-static member function`

To avoid the warning, use `static int f() ...`

--- **Note** ---

This option is not available when compiling with the `--old_cfe` option.

-Wu   Deprecated. For C code, -Wu suppresses warnings about future compatibility with C++. Warnings are suppressed by default. You can enable them with -W+u. For example:

`int *new(void *p) { return p; }`

results in the following warnings:

```
C2204W: C++ keyword used as identifier: 'new'
C2920W: implicit cast from (void *), C++ forbids
```

-Wv   Suppresses warning messages of the type:

`C2218W: implicit 'int' return type for 'f' - 'void' intended?`

This is usually caused by a return from a function that was assumed to return **int**, because no other type was specified, but is being used as a void function. This is widespread in old-style C. Such action always results in an error in C++.

-Wx   Suppresses unused declaration warnings such as:

`C2870W: variable 'y' declared but not used`

By default, unused declaration warnings are given for:

- local (within a function) declarations of variables, typedefs, and functions
- labels (always within a function)
- top-level static functions and static variables.

-Wy   Turns off warnings about deprecated features, for example:

C2067I: option -zas will not be supported in future releases of the
compiler

E.2.4   Specifying additional checks

The options described below give you control over the extent and rigor of the checks.
Additional checking is an aid to portability and is good coding practice:

-fa   Checks for certain types of data flow anomalies. The compiler performs
            data flow analysis as part of code generation. The checks indicate when
            an automatic variable might have been used before being assigned a
            value. The check is pessimistic and sometimes reports an anomaly where
            there is none. In general, it is useful at some stage to check all code using
            -fa.

-fh   Deprecated. Checks that:

- all external objects are declared before use

- all file-scoped static objects are used

- all predeclarations of static functions are used between their
  declaration and their definition. For example:
  ```
  static int f(void);
  static int f(void){return 1;}
  line 2: Warning: unused earlier static declaration of 'f'
  ```

- external objects declared only in included header files are used in a
  source file.

These checks directly support good modular programming practices.

When writing production software, use the -fh option only in the later
stages of program development. The extra diagnostics can be annoying in
the earlier stages.

-fp   Deprecated. Reports on explicit casts of integers to pointers, for example:

char *cp = (char *) anInteger;

This warning indicates potential portability problems. Casting explicitly between pointers and integers, although not clean, is not harmful on the ARM processor where both are 32-bit types. This option also causes casts to the same type to produce a warning. For example:

```
int f(int i) {return (int)i;}
        // Warning: explicit cast to same type.
```

-fv   Deprecated. Reports on all unused declarations (including from standard headers).

-fx   Deprecated. Enables all warnings normally suppressed by default, with the exception of the additional checks described in this section.

E.2.5   Error message options

The compiler issues errors to indicate serious problems in the code it is attempting to compile. The compiler options described below enable you to:

* turn off specific recoverable errors
* downgrade specific errors to warnings.

─────── **Caution** ───────

These options force the compiler to accept C and C++ source that normally produces errors. If you use any of these options to ignore error messages, it means that your source code does not conform to the appropriate C or C++ standard.

These options can be useful during development, or when importing source code from other environments. However, they might permit code to be produced that does not function correctly. It is generally better to correct the source than to use options to switch off error messages.

─────────────────

The general form of the -E compiler option is:

-E[*options*][+][*options*]

where *options* can be one or more of the letters a, c, or p as described below.

If the + character is included in the characters following the -E, the error messages corresponding to any following letters are enabled rather than suppressed.

─────── **Note** ───────

The -E option on its own without any options is the preprocessor switch. See *Specifying additional checks* on page E-9.

─────────────────

You can specify multiple options. For example:

-Eac

turns off the error messages specified by a and c.

The following error message options are available. Any message listed for an option is the old style message. To find the equivalent message displayed when you do not specify the --old_cfe option, see *Mapping of compiler messages* on page E-13.

-Ea For C++ only, this option downgrades access control errors to warnings. For example:

```
class A { void f() {}; };   // private member
A a;
void g() { a.f(); }         // erroneous access
C3032E: 'A::f' is a non-public member
```

———— **Note** ————

This option is not available when compiling with the --old_cfe option.

-Ec Suppresses all implicit cast errors, such as implicit casts of a nonzero **int** to **pointer**.

```
C3029E: '=': implicit cast of non-0 int to pointer
```

-Ef Deprecated. Suppresses errors for unclean casts, such as **short** to **pointer**.

-Ei Deprecated. For C++ only, this option downgrades from error to warning the use of implicit **int** in constructs such as const i;.

```
C2225W: declaration lacks type/storage-class (assuming 'int'): 'i'
```

———— **Note** ————

This option is not available when compiling with the --old_cfe option.

-El Deprecated. Suppresses errors about linkage disagreements where functions are implicitly declared as **extern** and then later redeclared as **static**.

```
C2991E: linkage disagreement for 'f' - treated as 'extern'
```

-Ep Suppresses errors arising as the result of extra characters at the end of a preprocessor line, for example:

```
C2839E: junk at end of #endif line - ignored
```

-Ez Deprecated. Suppresses the errors caused by zero-length arrays.

```
C2455E: array [0] found
```

E.3   Mapping old compiler options to the new options

This section maps the options and messages from older compilers, to the options and messages of the RVCT v2.0 compiler. It includes:

- *Mapping of compiler options*
- *Mapping of compiler messages* on page E-13.

E.3.1   Mapping of compiler options

Table E-1 shows the options to use in the RVCT v2.0 compiler that are equivalent to the older ARM compiler options.

Table E-1 Mapping of compiler options

Old compiler option	Equivalent new compiler option
`-ansi`	`--c90`
`-ansic`	`--c90`
`-auto_float_constants`	`--fpmode fast`
`-list`	`--list` ——— **Note** ——— Although the option name is the same, the behavior is different.
`-fa`	`-fa`
`-fh`	None
`-fi`	None
`-fj`	None
`-fp`	None
`-fu`	None
`-fv`	None
`-fx`	None

Table E-1 Mapping of compiler options (continued)

Old compiler option	Equivalent new compiler option
–zas*Number*	None
–W*letter*	--diag_suppress with the number of the message associated with the option *letter*. See *Mapping of compiler messages* for details.
–E*letter*	--diag_suppress with the number of the message associated with the option *letter*. See *Mapping of compiler messages* for details.

E.3.2 Mapping of compiler messages

For the older ARM compiler messages that are specified in this appendix, Table E-2 shows the equivalent messages that are output by the RVCT v2.0 compiler.

Table E-2 Mapping of compiler messages

Option	Old compiler message	New compiler message
–Wa	C2916W: Use of '=' in a condition context	#1293-D: assignment in condition
–Wb	C2485W: ANSI C forbids bit field type	#230: nonstandard type for a bit field
–Wd	C2215W: Deprecated declaration foo() - give arg types	#1295-D: Deprecated declaration foo - give arg types
–We	C2564W: extended constant initialiser used	None
–Wf	C2207W: Inventing 'extern int foo();'	#223-D: function declared implicitly
–Wg	C2819W: Header file not guarded against multiple inclusion	#1297-D: Header file not guarded against multiple inclusion
–Wi	C2887W: implicit constructor 'struct X'()	None
–Wk	C2621W: double constant automatically converted to float	None

Table E-2 Mapping of compiler messages (continued)

Option	Old compiler message	New compiler message
-Wl	C2915W: lower precision in wider context	None
-Wm	C2203W: non-portable - not 1 char in '...'	#69-D: integer conversion resulted in truncation
-Wn	C2921W: implicit narrowing cast	None
-Wo	C2201W: '*Number*' treated as '*Number*ll'	#1134-D: literal treated as "long long"
-Wp	C2812W: Non-ANSI #include <...>	None
-Wq	C2252W: base *class* will be initialized in declaration order instead of the order written C2253W: member *name* will be initialized in declaration order instead of the order written	#1299-D: members and base-classes will be initialized in declaration order, not in member initialisation list order
-Wr	C2997W: 'Derived::f()' inherits implicit virtual from 'Base::f()'	#1300-D: f inherits implicit virtual
-Ws	C2221W: padding inserted in struct 's'	#1301-D: padding inserted in struct X
-Wt	C2924W: 'this' unused in non-static member function	None
-Wu	C2204W: C++ keyword used as identifier: 'new' C2920W: implicit cast from (void *), C++ forbids	#40: expected an identifier
-Wv	C2218W: implicit 'int' return type for 'f' - 'void' intended?	#837-D: omission of explicit type is nonstandard ("int" assumed)
-Wx	C2870W: variable 'y' declared but not used	#177-D: variable "y" was declared but never referenced
-Wy	C2067I: option -zas will not be supported in future releases of the compiler	None
-Ea	C3032E: 'A::f' is a non-public member	#265-D: function "A::f" is inaccessible

Table E-2 Mapping of compiler messages (continued)

Option	Old compiler message	New compiler message
-Ec	C3029E: '=': implicit cast of non-0 int to pointer	#144: a value of type "int" cannot be used to initialize an entity of type "usertype *"
-Ei	C2225W: declaration lacks type/storage-class (assuming 'int'): 'i'	#77-D: this declaration has no storage class or type specifier
-El	C2991E: linkage disagreement for 'f' - treated as 'extern'	#172: external/internal linkage conflict with previous declaration
-Ep	C2839E junk at end of #endif line - ignored	C2839E junk at end of #endif line - ignored
-Ez	C2455E: array [0] found	#94: the size of an array must be greater than zero

Glossary

American National Standards Institute (ANSI)

An organization that specifies standards for, among other things, computer software. This is superseded by the International Standards Organization.

ATPE ARM Toolkit Proprietary ELF executable and linking format. ARM code generation tools produce objects and executable images in ATPE format.

Anachronisms Various C++ language features that are no longer strictly legal.

ANSI *See* American National Standards Institute.

API Application Program Interface.

Architecture The term used to identify a group of processors that have similar characteristics.

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.

ARM-Thumb Procedure Call Standard (ATPCS)

ARM-Thumb Procedure Call Standard defines how registers and the stack are to be used for subroutine calls.

ATPCS *See* ARM-Thumb Procedure Call Standard.

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.
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.
Double word	A 64-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
DWARF	Debug With Arbitrary Record 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.
Execution view	The address of regions and sections after the image has been loaded into memory and started execution.
Flash memory	Non-volatile 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, RealView Debugger).
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 Output sections.

See also Embedded assembler.

International Standards Organization
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.

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.

RVIC-ME
A multi-processor JTAG-based debug tool for embedded systems. Multi-ICE is an ARM registered trademark.

PCH
Precompiled Header

See also Precompiled Header.

PIC
Position Independent Code.

See also Read Only Position Independent.

PID
Position Independent Data.

See also Read Write Position Independent.

RealView Developer Kit (RVDK)

RealView Developer Kit is a suite of tools, together with supporting documentation and examples, that enable you to write, build and debug applications for the ARM architecture-based RISC processors.

Precompiled Header   A header file that is precompiled. This avoids the compiler having to compile the file each time it is included by source files.

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.

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.

Read Only Position Independent (ROPI)

Code and read-only data addresses can be changed at run-time.

Read Write Position Independent (RWPI)

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

ROPI   *See* Read Only Position Independent.

RTOS   Real Time Operating System.

RVDK   *See* RealView MCU Developer Kit.

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 programer-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.

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.

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.

Copyright © 2003, 2004 ARM Limited. All rights reserved.

Symbols

V

W