RealView™ Compilation Tools for BREW

Version 1.2

Compilers and Libraries Guide

ARM®

ARM DUI 0171B

RealView Compilation Tools for BREW
Compilers and Libraries Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
August 2001	A	Release 1.0
June 2002	B	Release 1.2

Proprietary Notice

Contents
RealView Compilation Tools for BREW Compilers and Libraries Guide

Preface

This preface introduces the *RealView Compilation Tools for BREW* v1.2 compilers and libraries reference documentation. It contains the following sections:

- *About this book* on page vi
- *Feedback* on page x.

About this book

This book provides reference information for RealView Compilation Tools for BREW (RVCT for BREW) v1.2. It describes the command-line options to the compilers. The book also gives reference material on the ARM implementation of the C and C++ compilers and the C libraries.

Intended audience

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

Using this book

This book is organized into the following chapters and appendixes:

Chapter 1 ***Introduction***

Read this chapter for an introduction to RVCT for BREW v1.2 compilers and libraries.

Chapter 2 ***C and C++ Compilers***

Read this chapter for an explanation of all command-line options accepted by the ARM C and C++ compilers.

Chapter 3 ***ARM Compiler Reference***

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

Chapter 4 ***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.

Chapter 5 ***Floating-point Support***

Read this chapter for a description of floating-point support in RVCT for BREW.

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 the ARM C and C++ compilers.

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 periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets and addenda, and the ARM Frequently Asked Questions.

ARM publications

This book contains reference information that is specific to development tools supplied with RVCT for BREW. Other publications included in the suite are:

- *RealView Compilation Tools for BREW v1.2 Installation Guide* (ARM DSI 0024)
- *RealView Compilation Tools for BREW v1.2 Assembler Guide* (ARM DUI 0170)
- *RealView Compilation Tools for BREW v1.2 Linker and Utilities Guide* (ARM DUI 0212).

The *RealView Compilation Tools for BREW v1.2 Assembler Guide* and *RealView Compilation Tools for BREW v1.2 Linker and Utilities Guide* are supplied in PDF format in `install_directory\PDF`.

For license management information, see the *ARM FLEXlm License Management Guide* (ARM DUI 0209). This is supplied in PDF format in `install_directory\PDF\ARM_FLEXlmGuide.pdf`.

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

- *ARM ELF specification* (SWS ESPC 0003)
- *TIS DWARF 2 specification*
- *ARM-Thumb Procedure Call Standard specification* (SWS ESPC 0002)
- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- the ARM datasheet or technical reference manual for your hardware device.

Other publications

This book is not intended to be an introduction to 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 book describes the C++ language:

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

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 Compilation Tools for BREW and the documentation.

Feedback on RealView Compilation Tools for BREW

If you have any problems with RealView Compilation Tools for BREW, 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 have any problems with 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 compilers, libraries, linker, and utility programs provided with RVCT for BREW. It contains the following sections:

1.1 About the compilers and libraries

RVCT for BREW consists of a suite of applications, together with supporting documentation, that enable you to write applications for the ARM family of RISC processors. You can use RVCT for BREW to build C, C++, and ARM assembly language programs.

The RVCT for BREW toolkit consists of the following major components:

- command-line development tools
- utilities
- supporting software.

This book describes the ARM compilers and libraries provided with RVCT for BREW. See *ARM publications* on page viii for a list of the other books in the RVCT for BREW documentation suite that give information on the ARM assembler, and supporting software.

1.2 The ARM compilers and libraries

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

1.2.1 The C and C++ compilers

RVCT for BREW provides the following compilers:

armcc The ARM C compiler. The compiler is tested against the Plum Hall C Validation Suite for ANSI conformance. It compiles ANSI C source into 32-bit ARM code.

armcpp This is the ARM C++ compiler. It compiles ANSI C++ or EC++ source into 32-bit ARM code.

tcc The Thumb® C compiler. The compiler is tested against the Plum Hall C Validation Suite for ANSI conformance. It compiles ANSI C source into 16-bit Thumb code.

tcpp This is the Thumb C++ compiler. It compiles ANSI C++ or EC++ source into 16-bit Thumb code.

The ARM compilers are optimizing compilers. Command-line options enable you to control the level of optimization.

The compilers generate output objects in ELF format, and generate DWARF2 debug information. In addition, the compilers can generate an assembly language listing of the output code, and can interleave an assembly language listing with source code.

See Chapter 2 *C and C++ Compilers* for more information on the ARM compilers.

1.2.2 The C and C++ libraries

RealView Compilation Tools for BREW provides the following runtime C and C++ libraries:

RealView Compilation Tools for BREW C libraries

RealView Compilation Tools for BREW C libraries provide standard C functions, and helper functions used by the C and C++ libraries.

Support libraries

RealView Compilation Tools for BREW 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 ANSI 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. See Chapter 4 *The C and C++ Libraries* for more information on the libraries.

1.3 Linking compiled objects

The ARM and Thumb linker combines the contents of one or more object files with selected parts of one or more object libraries to produce an ELF executable image, or a partially linked ELF object.

The linker can link ARM code and Thumb code, and automatically generates interworking veneers to switch processor state when required. The linker also automatically generates long branch veneers, where required, to extend the range of branch instructions.

The linker supports command-line options that enable you to specify the location of code and data in memory for simple images.

The linker can perform common section elimination and unused section elimination to reduce the size of your output image. In addition, the linker enables you to:

- produce debug and reference information about linked files
- generate static callgraph information
- control the contents of the symbol table in output images.

The linker automatically selects the appropriate standard C or C++ library variants to link with, based on the build attributes of the objects it is linking.

The linker does not generate output formats other than ELF. To convert ELF images to other format, such as plain binary for loading into ROM, use the fromELF utility. See *Related utilities* on page 1-6.

See *RealView Compilation Tools for BREW Linker and Utilities Guide* for detailed information on the ARM linker.

1.4 Related utilities

This section gives an overview of the fromELF utility tool that is provided to support the main development tools. See *RealView Compilation Tools for BREW Linker and Utilities Guide* for detailed information on this utility.

1.4.1 fromELF

fromELF is the ARM image conversion utility. It accepts ELF format input files and converts them to a variety of output formats, including:

- plain binary
- Motorola 32-bit S-record format
- Intel Hex-32 format
- Byte Oriented (Verilog Memory Model) Hex format
- *Extended Intellec Hex* (IHF) format (this option is deprecated and will be removed from future versions of the product).

The utility can also:

- produce textual information about the input file
- disassemble code
- resave in ELF format.

Chapter 2
C and C++ Compilers

This chapter describes the command-line options to the ARM and Thumb, C and C++ compilers. This chapter assumes you are familiar with command-line software development tools such as those provided with RVCT for BREW. It contains the following sections:

- *About the C and C++ compilers* on page 2-2
- *File usage* on page 2-5
- *Command syntax* on page 2-9.

2.1 About the C and C++ compilers

Wherever possible, the compilers adopt widely used command-line options familiar to users of Windows/MS-DOS.

The ARM C compilers compile C as defined by ISO/IEC 9899:1990 (E) including its Technical Corrigendums 1 & 2. Some features of C99, such as long long, are also available.

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

The ARM C++ compilers can also compile the subset of standard C++ known as *Embedded C++* (EC++). EC++ is a subset of standard C++ that provides efficient code for use in embedded systems. The EC++ amendment to the ISO standard is evolving. The proposed definition is available on the web at `http://www.caravan.net`.

Note

The compilers do not support the additions to C90 in Normative Addendum 1, that is `wchar.h` and `wctype.h`, added in 1994.

2.1.1 Compiler variants

All ARM C and C++ compilers accept the same basic command-line options. Unless stated otherwise, the text in this chapter applies to all compiler types. Where a specific compiler has added features or restrictions, this is noted in the text. Where an option applies only to C++, this is also noted in the text.

The compiler variants are shown in Table 2-1.

Table 2-1 Compiler variants

Compiler name	Compiler variant	Source language	Compiler output
armcc	C	C	32-bit ARM code
tcc	C	C	16-bit Thumb code
armcpp	C++	C or C++	32-bit ARM code
tcpp	C++	C or C++	16-bit Thumb code

Note

Throughout this chapter, the phrase *the ARM compilers* refers to armcc, armcpp, tcc, and tcpp.

2.1.2 Source language modes

The ARM compilers have three distinct source language modes that you can use to compile several varieties of C and C++ source code:

ANSI C In ANSI C mode, the ARM compilers pass release 7.00 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.

EC++ This mode applies only to the ARM C++ compilers. The ARM C++ compilers compile the Embedded C++ subset of the ISO/IEC Standard C++.

C++ This mode applies only to the ARM C++ compilers. The ARM C++ compilers compile ISO/IEC standard C++. The compilers are tested against *Suite++, The Plum Hall Validation Suite for C++, version 5.00*. This is the default language mode for the ARM C++ compilers. 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 2-14.

2.1.3 Library support

RVCT for BREW provides ANSI C libraries in prebuilt binary form. See Chapter 4 *The C and C++ Libraries* for detailed information about the libraries.

2.2 File usage

This section describes naming conventions and included files.

2.2.1 Naming conventions

The ARM compilers use 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 compilers use or generate files with the following file suffixes:

filename`.c` — ARM C compilers recognize the `.c` suffix as source files.

ARM C++ compilers recognize `.c`, `.cpp`, `.cp`, `.c++`, and `.cc` suffixes as source files.

filename`.h` — Header file (a convention only, this suffix has no special significance for the compiler).

filename`.o` — ARM object file in ELF format.

filename`.s` — ARM or Thumb assembly language file. (This can be placed in the input file list or, with the `-S` option, produced as an output file from the C and C++ compilers.)

filename`.lst` — Error and warning list file (the default output extension for `-list` option).

Filename validity

The compilers do not check that filenames are acceptable. If a filename is not acceptable, the compiler reports that the file cannot be opened, but the compiler gives no more diagnosis.

Output files

By default, the output files created by an ARM compiler are stored in the current directory. Object files are written in *ARM Executable and Linkable Format* (ELF). The ELF documentation is available in *install_directory*`\PDF`.

2.2.2 Included files

Several factors affect the way the ARM compilers search 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 *ARMINC*
- whether the filename is an absolute filename or a relative filename
- whether the filename is between angle brackets or double quotes.

The in-memory file system

The ARM compilers have the ANSI C library headers built into a special, textually-compressed, in-memory file system. By default, the C header files are used from this file system for applications built from the command line. You can specify the in-memory file system on the command line with `-j-` or `-I-`.

The C++ header files that are equivalent to the C library header files are also stored in the in-memory file system. The header files specific to C++, such as `iostream`, are not stored in the in-memory file system.

Enclosing a filename in angle brackets, `#include <stdio.h>` for example, indicates that the included file is a system file and instructs the compiler to look in the in-memory file system first.

Enclosing a filename in double quotes, `#include "myfile.h"` for example, indicates that it is not a system file and instructs the compiler to look in the search path.

The current place

By default, the ARM compilers search for source files and `#include` header files 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 *`install_directory`*`\include` and the compiler is seeking the include file `sys\defs.h`, it locates *`install_directory`*`\include\sys\defs.h` if it exists.

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

The original current place `install_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 ARMINC environment variable

You can set the `ARMINC` environment variable to a comma-separated list of directories to control searching for included header and source files. For example, from a Windows command line, type:

```
set ARMINC=c:\work\x,c:\work\y
```

When compiling from the command line, directories specified with `ARMINC` are searched immediately after directories specified by the `-I` option on the command line have been searched. If the `-j` option is used, `ARMINC` is ignored.

The search path

Table 2-2 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:

`:mem` The in-memory file system where the ARM compilers store ANSI C and some C++ header files. See *The in-memory file system* on page 2-6 for more information.

`ARMINC` The list of directories specified by the `ARMINC` environment variable, if it is set.

`CP` The current place. See *The current place* on page 2-6 for more information.

`Idir and jdirs`

The directories specified by the `-I` and `-j` compiler options.

Table 2-2 Include file search paths

Compiler option	<include>	"include"
Neither `-I` or `-j`	`:mem` and `ARMINC`	`CP`, `ARMINC`, and `:mem`
`-j`	`jdirs`	`CP` and `jdirs`
`-I`	`:mem`, `ARMINC`, and `Idirs`	`CP`, `Idirs`, `ARMINC`, and `:mem`
Both `-I` and `-j`	`Idirs` and `jdirs`	`CP`, `Idirs`, and `jdirs`
`-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

2.3 Command syntax

This section describes the command syntax for the ARM C and C++ compilers.

You can control many aspects of compiler operation with command-line options. All options are prefixed by a minus – sign, and some options are followed by an argument. In most cases the ARM C and C++ compilers permit space between the option letter and the argument.

2.3.1 Invoking the compiler

The command for invoking the ARM compilers is:

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

The command-line options can appear in any order. The options are:

compiler This is one of armcc, tcc, armcpp, or tcpp.

PCS-options This specifies the procedure call standard to use. See *Procedure Call Standard options* on page 2-12 for details.

source-language This specifies the variant of source language that is accepted by the compiler. The default is ANSI C for the C compilers and ISO Standard C++ for the C++ compilers. See *Setting the source language* on page 2-14 for details.

search-paths This specifies the directories that are searched for included files. See *Specifying search paths* on page 2-15 for details.

preprocessor-options

This specifies preprocessor behavior, including preprocessor output and macro definitions. See *Setting preprocessor options* on page 2-15 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 2-17 for details.

target-options This specifies the target processor or architecture. See *Specifying the target processor or architecture* on page 2-19 for details.

debug-options This specifies whether or not debug tables are generated, and their format. See *Generating debug information* on page 2-19 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 2-21 for details.

warning-options This specifies whether specific warning messages are generated. See *Controlling warning messages* on page 2-26 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 *Specifying additional checks* on page 2-30 for details.

error-options This enables you to turn off specific recoverable errors or downgrade specific errors to warnings. See *Controlling error messages* on page 2-32 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 assembler activates to process the source file.

Reading compiler options from a file

When the operating system restricts the command line length, use the following option to read additional command-line options from a file:

`-via` *filename*

This opens a file and reads additional command-line options from it. You can nest `-via` calls within via files by including `-via` *filename2* in the file.

In the following example, the options specified in `input.txt` are read as the command-line is parsed:

```
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`.

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.

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 RealView Compilation Tools for BREW compilers.

See the ATPCS specification for more information on the ARM and Thumb procedure call standards. See *Controlling code generation* on page 2-21 for other build options.

Use the following command-line options 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 for all compilers is:

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

The qualifiers are listed below.

Interworking qualifiers

`/nointerwork` This option generates code with no ARM/Thumb interworking support.

`/interwork` This option generates code with ARM/Thumb interworking support. See *RealView Compilation Tools for BREW Linker and Utilities Guide* for information on the automatically generated interworking veneers.

Position independence qualifiers

`/ropi` This option generates (read-only) position-independent code. This is the default and only option. The compiler:

- addresses read-only code and data pc-relative
- sets the *Position Independent* (PI) attribute on read-only output sections.

Note

The ARM tools cannot determine if the final output image will be *Read-Only Position Independent* (ROPI) until the linker finishes processing input sections. This means that the linker might emit ROPI error messages, even though you have selected this option.

`/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). `/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

The compiler does not force your read/write data to be position-independent. This means that the linker might emit RWPI warning messages, even though you have selected this option.

Stack checking qualifiers

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

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 2-21).

The following options specify how strictly the compiler enforces the standards and conventions of that language. By default, the C compilers compile ANSI-C, and the C++ compilers compile as much as they can of ISO/IEC C++.

`-ansi` This option compiles ANSI standard C. This is the default for armcc and tcc. The default mode is a fairly strict ANSI compiler, but without some of the inconvenient features of the ANSI standard. There are also some minor extensions allowed (for example `//` in comments and `$` in identifiers).

`-ansic` This option compiles ANSI standard C. This option is synonymous with the `-ansi` option.

`-cpp` This option compiles ISO/IEC C++. This option is the default with the C++ compilers and not available with the C compilers.

`-strict` This option enforces more stringent conformance to the C standard ISO/IEC 9899:1990 (E) and the C++ standard ISO/IEC 14882:1998 (E). For example:

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

This gives an error when compiled with `-cpp -strict`, but only a warning with `-cpp`. Because no object is declared, `static` is spurious. In the C++ standard, the code shown is therefore illegal.

You can combine language options:

`armcc -ansi` Compiles ANSI standard C. This is the default.

`armcc -strict` Compiles strict ANSI standard C.

`armcpp` Compiles standard C++.

`armcpp -ansi` Compiles normal ANSI standard C (C mode of C++).

`armcpp -ansi -strict`

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

`armcpp -strict` Compiles strict C++.

2.3.4 Specifying search paths

The following options specify the directories that are searched for included files.

The precise search path varies according to the combination of options selected and whether the include file is enclosed in angle brackets or double quotes. See *Included files* on page 2-6 for full details of how these options work together.

`-I, dir-name` This option adds the specified directory (or comma-separated list of directories) to the list of places that are searched for included files. If you specify more than one directory, the directories are searched in the same order as the `-I` options specifying them.

RVCT for BREW compilers use an in-memory file system to speed processing of include header files. The in-memory file system is specified by `-I-`.

`-fk` This option uses Kernighan and Ritchie search rules for locating included files. The current place is defined by the original source file and is not stacked. See *The current place* on page 2-6 for more information. If you do not use this option, Berkeley-style searching is used.

`-fd` This option makes the handling of quoted include files the same as angle-bracketed include files. Specifically, the current place is excluded from the search path.

`-jdir-list`

This option adds the specified comma-separated list of directories to the end of the search path after all the directories specified by the `-I` options. Use `-j-` to search the in-memory file system.

2.3.5 Setting preprocessor options

The following command-line options control aspects of the preprocessor. (See *Pragmas* on page 3-2 for descriptions of other preprocessor options that can be set by pragmas.)

`-E` This option 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:

```
compiler-name -E source.c > raw.c
```

You can also use the `-o` option to specify a file for the preprocessed output. 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` This option retains comments in preprocessor output when used in conjunction with `-E`. This option differs from the `-c` (lowercase) option that suppresses the link step. See *Specifying output format* on page 2-17 for a description of the `-c` option.

`-D, symbol=value`

This option 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.

If you use a quoted string value, for example `-DFOO_VALUE="FOO_VALUE"`, you must delimit the quotes as follows:

- from the command line, use `-DFOO_VALUE=\"FOO_VALUE\"`
- in a via file, use `'-DFOO_VALUE="FOO_VALUE"'` or `-DFOO_VALUE='"FOO_VALUE"'`.

`-Dsymbol`

This option 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.

`-M` This option executes only the preprocessor phase of the compiler, as with `-E`. 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:

```
compiler-name -M source.c >> Makefile
```

If you specify the `-o filename` option, the dependency lines generated on standard output refer to `filename.o`, not to `source.o`. However, no object file is produced with the combination of `-M -o filename`.

`-Usymbol`

This option undefines *symbol*. This has the same effect as the text `#undef symbol` at the head of the source file. You can repeat this option.

2.3.6 Specifying output format

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

Use the following options to direct the compiler to create unlinked object files, assembly language files, or listing files from C or C++ source files.

`-c` This option compiles but does not perform the link step. The compiler compiles the source program and writes the object files to either the current directory or the file specified by the `-o` option. This option is different from the uppercase `-C` option, described in *Setting preprocessor options* on page 2-15. (The `-C` option retains comments in preprocessor output.)

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

Caution

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` This option is used with `-list` to list the lines from any files included with directives of the form `#include "file"`.

`-fj` This option is used with `-list` to list the lines from any files included with directives of the form `#include <file>`.

`-fu` This option is 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;
```

`-o` *`file`*

This option 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.

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

`-depend` *`filename`*

This option is the same as `-MD`, but writes makefile dependency lines to the specified file.

`-S` This option 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.c` is the name of the source file stripped of any leading directory names. The default file name can be overridden with the `-o` option.

Note

You can use armasm to assemble the output file and produce object code. The compilers add `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`.

2.3.7 Specifying the target processor or architecture

The options described in this section specify the target processor or architecture attributes for a compilation.

Note

The RealView Compilation Tools for BREW compilers support the ARM7TDMI processor only.

The following general points apply to processor and architecture options:

- Use only a single processor or architecture name with `-cpu`. You cannot specify both a processor and an architecture.
- Specifying a Thumb-aware processor, such as `-cpu ARM7TDMI` to armcc or armcpp does not make these compilers generate Thumb code. It only allows features of the processor to be used, such as interworking instructions. Use tcc or tcpp to generate Thumb code.

The following options are available:

`-cpu` *name*
: *name* can be either the processor name (ARM7TDMI), or the architecture name (4T). You must enter the name exactly. Wildcard characters are not accepted.

`-fpu` *name*
: This option selects the target *Floating-Point Unit* (FPU) architecture.

  Valid options 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.

2.3.8 Generating debug information

This section describes options that enable you to specify whether debug tables are generated for the current compilation and, if they are, specify their format. 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 2-21 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 `-g` is used or is not used. The only difference is the existence of debug tables.

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

```
-g -dwarf2 -O0 -gt+p
```

`-g+` is a synonym for `-g`.

`-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 compilers:

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

2.3.9 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*
- *Controlling code and data sections* on page 2-25
- *Setting byte order* on page 2-25

Defining optimization criteria

The following options control aspects of how the compilers optimize generated code.

`-O`, *`number`* This option 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` This option optimizes 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` This option optimizes 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:

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

This is compiled 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.

`-Ono_inline` This option disables inlining of functions. Calls to inline functions are not expanded inline. You can use this option to help debug inline functions.

`-Oinline` This option enables the compiler to inline functions. This is the default.

The compiler inlines functions when it is sensible to do so:

- Automatically, for optimization level `O2` unless the `-Ono_autoinline` option is specified.
- When the function is qualified as an inline function, for example with the `__inline` keyword in C or the `inline` keyword in C++. This applies for all optimization levels. Functions qualified as inline functions are more likely to be inlined, but the qualifier is only a hint to the compiler. See *Function keywords* on page 3-6.

The compiler changes its criteria for inlining functions depending on whether you select `-Ospace` or `-Otime`. Selecting `-Otime` increases the number of functions that are inlined.

Note

Setting breakpoints in ROM images

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

`-Ono_autoinline`

This option disables automatic inlining. This is the default for optimization levels `-O1` and `-O0` if `-Oinline` is enabled.

`-Oautoinline` This option enables automatic inlining. It 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.

`-Ono_data_reorder`

This option disables automatic reordering of top-level data items (globals, for example). The C/C++ compilers save memory by eliminating wasted space between data items. However, this optimization can break legacy code, if the code (incorrectly) makes 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.

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

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

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.

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.

Setting the default type of unqualified floating-point constants

`-auto_float_constants`

This option changes the type of unsuffixed floating-point constants from **`double`** (as specified by the ANSI/ISO C and C++ standards) to *unspecified*. In this context, *unspecified* means that uncast **`double`** constants and **`double`** constant expressions are treated as **`float`** when used in expressions with values other than **`double`**. This can sometimes improve the execution speed of a program that uses **`float`** variables.

Compile-time evaluation of constant expressions that contain such constants is unchanged. The compiler uses double-precision calculations, but the unspecified type is preserved. For example:

```
(1.0 + 1.0) // evaluates to the floating-point
            // constant 2.0 of double precision and
            // unspecified type.
```

In a binary expression that must be evaluated at runtime (including expressions that use the ?: operator), a constant of unspecified type is converted to **`float`**, instead of **`double`**. The compiler issues the following warning:

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

You can avoid this warning by explicitly suffixing floating-point constants that you want to be treated as **`float`** with an `f` as shown in Example 2-1. You can turn this warning off with the `-Wk` compiler option.

Note

This behavior is not in accordance with the ANSI C standard.

If the other operand in the expression has type **`double`**, a constant of unspecified type is converted to **`double`**. A cast of a constant of unspecified type to type T produces a constant of type T (Example 2-1).

Example 2-1 Double and float

```
float f1(float x) { return x + 1.0; }  // Uses float add and is treated the same
                                       // as f2() below, a warning is issued.

float f2(float x) { return x + 1.0f;}  // Uses float add with no warning, with
                                       // or without -auto_float_constants.

float f3(double x) { return x + 1.0;}  // Uses double add,
                                       // no special treatment.
```

```
float f4(float x) { return x + (double)1.0;}  // Uses double add,
                                              // no special treatment.
```

Controlling code and data sections

`-zo` This option generates one ELF section for each function in 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 enables the linker to remove unused functions when the default `-remove` linker option is active. 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, when creating code for a library, it can prevent unused functions being included at the link stage. This can result in the reduction of the final image size. If you are using third-party code, you do not have to change the source, but you must recompile (unless the code was already compiled with the -zo option).

pragma arm section

This pragma specifies the code or data section name used for subsequent functions or objects. This includes definitions of anonymous objects the compiler creates for initializations.

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 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 lowest address.

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, ARM compilers assume that the memory system can load and store words at four-byte alignment, halfwords at two-byte alignment, and bytes. Load and store capability can be indicated by specifying *`option`*:

`-S22` The memory cannot store halfwords. You can use this to suppress the generation of `STRH` instructions when generating ARM code for the architecture v4T processor.

`-L22` The memory cannot load halfwords. You can use this to suppress the generation of `LDRH` instructions when generating ARM code for the architecture v4T processor.

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.

2.3.10 Controlling warning messages

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 2-30 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 warning message options are as follows:

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

-Wa
: This option suppresses the warning:

```
C2961W: 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
: This option suppresses the warning messages that are issued for extensions to the ANSI standard. Examples include:

- using an unwidened type in an ANSI 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
: This option 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 ANSI C mode.

In ANSI 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
: This option suppresses the warning messages given when an extended constant expression is used in an initializer (see *C language extensions* on page 3-16) that other C compilers are not required by the standard to accept.

-Wf
: This option suppresses the message:

```
Inventing extern int foo()
```

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

`-Wg`
: This option 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`
: This option 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`.

`-Wl`
: This option turns off the warning:

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

when code like 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`
: This option suppresses warnings about multiple-character **`char`** constants:

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

`-Wn`
: This option suppresses the warning message:

```
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` This option suppresses warnings for implicit conversion to signed **`long long`** constants.

`-Wp` This option suppresses the warning message:

```
C2812W: Non-ANSI #include <…>
```

The ANSI C standard requires that you use `#include <…>` for ANSI 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` This option suppresses warnings in C++ constructor initialization order.

`-Wr` This option 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.

`-Ws` This option 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` This option 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() ...`

`-Wu` 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` This option 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` This option 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` This option turns off warnings about deprecated features.

2.3.11 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` This 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`.

`-fh` This option 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` This option 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` This option reports on all unused declarations (including from standard headers).

`-fx` This option enables all warnings normally suppressed by default, with the exception of the additional checks described in this section.

2.3.12 Controlling error messages

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`* is a set of one or more of the letters `a`, `c`, `f`, `i`, `l`, `p`, or `z` 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 *Setting preprocessor options* on page 2-15.

You can specify multiple options. For example:

```
-Eac
```

turns off the error messages specified by `a` and `c`.

The following options are on by default:

`-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
```


`-Ec` This option 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` This option suppresses errors for unclean casts, such as **`short`** to **`pointer`**.

`-Ei` 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'
```

`-El` This option 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` This option suppresses errors arising as the result of extra characters at the end of a preprocessor line.

`-Ez` This option suppresses the errors caused by zero-length arrays.

```
C3017E: size of a [] array required, treated as [1]
```


Chapter 3
ARM Compiler Reference

This chapter gives information on ARM-specific features of the ARM C and C++ compilers. It contains the following sections:

For additional reference material on the ARM C and C++ compilers see also:

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

3.1 Compiler-specific features

This section describes the ARM-specific aspects of the ARM C and C++ compilers, including:

- *Pragmas*
- *Function keywords* on page 3-6
- *Variable declaration keywords* on page 3-9.

Note

Features described here are outside the ANSI 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
```

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

Table 3-1 Pragmas recognized by the ARM compilers

Pragma name	Default	Reference
`arm section`	Off	*Pragmas controlling code generation* on page 3-4
`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
`debug`	On	*Pragmas controlling debugging* on page 3-3
`import`	–	*Pragmas controlling code generation* on page 3-4
`Ospace`	–	*Pragmas controlling optimization* on page 3-3
`Otime`	–	*Pragmas controlling optimization* on page 3-3
`O`*`num`*	–	*Pragmas controlling optimization* on page 3-3

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 (uppercase O).

`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 2-21 for more information on optimization levels.

Pragmas controlling code generation

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

`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.

`arm section` *`section_sort_list`*

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

- declarations
- inline functions (and their local static variables)
- template instantiations (and their local static variables)
- elimination of unused variables and functions
- the order in which 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 on page 3-5.

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
```


3.1.2 Function keywords

Several keywords tell the compiler to give a function special treatment. These are all ARM extensions to the ANSI C specification:

Declarations inside functions

Declarations inside a function indicate that the following statements are processed differently.

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.

`__irq` This 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 in tcc or tcpp. No arguments or return values can be used with `__irq` functions.

`__pure` This 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 */
```

`__swi`

This 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`

This 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`

This 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.

`__inline` This 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 2-21 for information on command-line options that affect inlining.

`__weak` This specifies an **`extern`** function or object declaration that, if not present, does not cause the linker to fault an unresolved reference. The linker does not load the function or object from a library unless another compilation uses the function or object non-weakly. If the reference remains unresolved, its value is assumed to be `NULL`. See *RealView Compilation Tools for BREW Linker and Utilities Guide* for details on library searching.

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 non-weakly 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()` non-weakly from `h()`:

```
__weak void f(void);
void h() {f();}
void f() {}
```

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 3-12 for information on qualifiers such as **`volatile`** and `__packed`

Standard keywords

These keywords declare a storage class.

`register` Using the ARM compilers, 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 allowed.

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()`
: The `__align()` storage class modifier aligns a top-level object on a byte boundary.

`__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 RealView Compilation Tools for BREW compilers and C libraries ensure that eight-byte alignment of the stack is maintained.

`__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:

- floating-point types
- structures or unions with floating-point fields
- 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.

The effect of casting away __packed is undefined. The effect of casting a nonpacked 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).

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 on page 3-14).

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 ANSI 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 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 also *Function storage class modifiers* on page 3-8.

3.2 Language extensions

This section describes the language extensions supported by the ARM compilers.

3.2.1 C language extensions

The compilers support the ANSI C language extensions described below and in *C and C++ language extensions*. The extensions are not available if the compiler is restricted to compiling strict ANSI 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 `//`

constant expressions

Extended constant expressions, such as the following, are allowed in initializers:

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

3.2.2 C and C++ language extensions

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

Identifiers

The `$` character is a legal character in identifiers.

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 C and C++ compilers support 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
- an `llu` (or `ull`) 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 ANSI 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
```

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`**.

Keywords

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

- *Function keywords* on page 3-6
- *Variable declaration keywords* on page 3-9
- *Type qualifiers* on page 3-12.

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 `/ropi` or `/rwpi`. If you recompile existing code with either of these options, you must change the linkage specification for external constants that are dynamically initialized or have mutable members.

Compiling C++ with either the `/ropi` or `/rwpi` options 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 allowed 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.

3.3 C and C++ implementation details

This section describes implementation details for RealView Compilation Tools for BREW compilers, including:

- *Character sets and identifiers*
- *Basic data types* on page 3-22
- *Operations on basic data types* on page 3-23
- *Structures, unions, enumerations, and bitfields* on page 3-25.

3.3.1 Character sets and identifiers

The following points apply to the character sets and identifiers expected by the compilers:

- 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.
- 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.
- 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.
- RealView Compilation Tools for BREW compilers do not support 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. RealView Compilation Tools for BREW 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`.
- 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
`\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 RealView Compilation Tools for BREW C and C++.

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 3-25.)

- Local variables are usually kept in registers, but when local variables are spilled 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)
`long`	32	4 (word-aligned)
`long long`	64	4 (word-aligned)
`float`	32	4 (word-aligned)
`double`	64	4 (word-aligned)
`long double`	64	4 (word-aligned)
All pointers	32	4 (word-aligned)
`bool` (C++ only)	32	4 (word-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 3-24 for more information.

ARM implements an ANSI extension for floating-point constants (see *Hexadecimal floating-point constants* on page 3-18).

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

RealView Compilation Tools for BREW compilers perform 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 will 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.

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.

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.

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 nonpacked 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 3-29 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, nonpacked 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.

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`**.

```
struct {char c; int x; short s} ex1;
```

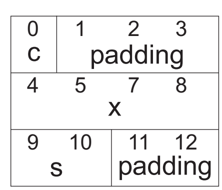

Figure 3-1 Conventional structure example

- Structures with empty initializers are allowed 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. Floating-point types cannot be fields of packed structures.

Packed structures are defined with the `__packed` qualifier. (See *ARM-specific keywords* on page 3-10.) There is no command-line option to change the default packing of structures.

Bitfields

In nonpacked structures, the ARM compilers allocate 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;
};
```

Bit number			
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0			
unallocated	y	padding	x

Figure 3-2 Bitfield allocation 1

——— **Note** ———

The same basic rules apply to bitfield declarations with different container types. For example, adding an **`int`** bitfield to the example 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).

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 non-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`.

3.4 Predefined macros

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

Table 3-4 Predefined macros

Name	Value	When defined
`__arm`	–	If using armcc, tcc, armcpp, or tcpp.
`__ARMCC_VERSION`	*ver*	For giving the version number of the compiler. The value is the same for the ARM and Thumb compilers. It is a decimal number, whose value can be relied on to increase between releases. The format is *PVtbbb* where: *P* is the product (1 for RVCT for BREW) *V* is the minor version (2 for 1.2) *t* is the patch release (0 for 1.2) *bbb* is the build (750 for example). The example given results in 120750.
`__APCS_INTERWORK`	–	If `-apcs /interwork` in use.
`__APCS_ROPI`	–	If `-apcs /ropi` in use.
`__BIG_ENDIAN`	–	If compiling for a big-endian target.
`__cplusplus`	–	In C++ compiler mode.
`__CC_ARM`	–	Returns compiler name.
`__DATE__`	*date*	When date of translation of source file is required.
`__FILE__`	*name*	The presumed full pathname of the current source file.
`__func__`	*name*	The name of the current function.
`__LINE__`	*num*	When line number of the current source file is required.
`__MODULE__`	*mod*	Contains the filename part of the value of `__FILE__`.
`__OPTIMISE_SPACE`	–	If `-Ospace` in use.
`__OPTIMISE_TIME`	–	If `-Otime` in use.
`__prettyfunc__`	*name*	The unmangled name of the current function.
`__sizeof_int`	4	For `sizeof(int)`, but available in preprocessor expressions.

Table 3-4 Predefined macros (continued)

Name	Value	When defined
`__sizeof_long`	4	For `sizeof(long)`, but available in preprocessor expressions.
`__sizeof_ptr`	4	`For sizeof(void *)`, but available in preprocessor expressions.
`__STDC__`	–	In all compiler modes.
`__STDC_VERSION__`	–	Standard version information.
`__STRICT_ANSI__`	–	Set by `-strict`.
`__TARGET_ARCH_4T`	–	For the compiler options `-cpu 4T` or `-cpu ARM7TDMI`.
`__TARGET_CPU_ARM7TDMI`	–	For the compiler option `-cpu ARM7TDMI`.
`__TARGET_FEATURE_HALFWORD`	–	If the target architecture supports halfword and signed byte access instructions.
`__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.
`__TARGET_FPU_`*xx*	–	Identifies the FPU option as one of: • `__TARGET_FPU_NONE` • `__TARGET_FPU_SOFTVFP` See the description of the `-fpu` option in *Specifying the target processor or architecture* on page 2-19 for more information on FPU options.
`__thumb`	–	If using tcc or tcpp.
`__TIME__`	*time*	When time of translation of the source file is required.

Chapter 4
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:

4.1 About the runtime libraries

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

ANSI C The C libraries consist of:

- the functions defined by the ISO C library standard
- helper functions used by the C and C++ compilers.

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

- helper functions for the C++ compiler
- additional C++ functions.

For a detailed description of how the libraries comply with the ISO standard, see *ISO implementation definition* on page 4-5.

4.1.1 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 floating point, or none.

Position-independence

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

When you build your application you must make certain fundamental choices that affect the compatibility of object code built by the RealView Compilation Tools for BREW tools. There is a variant of the ANSI C library for each combination of major build options. Build options are described in more detail in:

- the libraries chapter in *RealView Compilation Tools for BREW Linker and Utilities Guide*
- *Procedure Call Standard options* on page 2-12 for the compiler
- *RealView Compilation Tools for BREW Assembler Guide* for the assembler.

4.1.2 Library directory structure

The libraries are installed in `install_directory\lib\armlib`. This directory contains the variants of the RealView Compilation Tools for BREW C library, the floating-point arithmetic library, and the math library. The accompanying header files are in `install_directory\include`.

The environment variable `ARMLIB` must be set to point to the `lib` directory. Alternatively use the `-libpath` argument to the linker to identify the directory holding the library subdirectories. You do not have to identify `armlib` and `cpplib` separately. The linker finds them for you from the location of `lib`.

Note

- RealView Compilation Tools for BREW C libraries are supplied in binary form only.
- You must not modify the RealView Compilation Tools for BREW C libraries.

4.1.3 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 4-1). You must not use these, or other similar functions, in a reentrant application unless you build it using `-apcs /rwpi`.

Table 4-1 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 for BREW.

4.2 ISO implementation definition

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

4.2.1 ANSI C library implementation definition

The ANSI C library variants are listed in *Library naming conventions* on page 4-19.

The ANSI 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 4-2, when supplied with out-of-range arguments, respond in the way shown.

Table 4-2 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 noninteger`	`QNaN`	`EDOM`
`pow(x,y)`	`x<0, y noninteger`	`QNaN`	`EDOM`

Table 4-2 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 4-2 on page 4-6, all functions return `QNaN` when passed `QNaN` and throw an invalid operation exception when passed `SNaN`.

Signal function

The signals listed in Table 4-3 are supported by the `signal()` function.

Table 4-3 Signal function signals

Signal	Number	Description	Additional argument
SIGABRT	1	Abort	None
SIGFPE	2	Arithmetic exception	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	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	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++	-
SIGCPPL	13	Exception from C++	-
reserved	14-31	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.

For all the signals in Table 4-3 on page 4-8, 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 ANSI C standard.
- A zero-length file, into which no characters have been written by an output stream, does exist.
- 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 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 4-4.

Table 4-4 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 ANSI-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()`.

4.2.2 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 4-6 on page 4-12.

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 4-5.

Table 4-5 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 4-6.

Table 4-6 Standard C++ library differences

Standard	Implementation differences
Wide character	Not a separate type. `wchar_t` is an implicit typedef for **`unsigned short`**. Characters are 8-bits wide.
Namespaces	Not supported. All top-level items are in the global namespace.
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.

4.3 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 4-7. The headers `<stdint.h>` and `<inttypes.h>` from C99 are also available.

Table 4-7 Extensions

Function	Header file definition	Extension
atoll()	`stdlib.h`	C99 standard
strtoll() on page 4-14	`stdlib.h`	C99 standard
strtoull() on page 4-14	`stdlib.h`	C99 standard
snprintf() on page 4-14	`stdio.h`	C99 standard
vsnprintf() on page 4-15	`stdio.h`	C99 standard
lldiv() on page 4-15	`stdlib.h`	C99 standard
llabs() on page 4-15	`stdlib.h`	C99 standard
alloca() on page 4-16	`alloca.h`	C99 and others
_fisatty() on page 4-16	`stdio.h`	ARM-specific
__heapstats() on page 4-16	`stdlib.h`	ARM-specific
__heapvalid() on page 4-18	`stdlib.h`	ARM-specific

4.3.1 atoll()

The `atoll()` function converts a decimal string into an integer, similarly to the ANSI 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)
```

4.3.2 strtoll()

The `strtoll()` function converts a string in an arbitrary base to an integer, similar to the ANSI function `strtol()`, but returning a **`long long`** result. Like `strtol()`, the parameter *`endptr`* can point to a location in which to store a pointer to the end of the translated string, 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)
```

4.3.3 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)
```

4.3.4 snprintf()

`snprintf()` works almost exactly like the ANSI `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 ANSI header file, but the function is not allowed by the ANSI C library standard. It is therefore not available if you use the compilers with the `-strict` option.

Syntax

```
int snprintf(char *buffer, size_t bufsize,
             const char *format,  ...)
```

4.3.5 vsnprintf()

`vsnprintf()` works almost exactly like the ANSI `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 ANSI header file, but the function is not allowed by the ANSI C library standard. It is therefore not available if you use the compilers with the `-strict` option.

Syntax

```
int vsnprintf(char *buffer, size_t bufsize,
              const char *format, va_list ap)
```

4.3.6 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 ANSI function `ldiv`. The return type `lldiv_t` is a structure containing two **`long long`** members, called `quot` and `rem`.

`<stdlib.h>` is an ANSI header file, but the function is not allowed by the ANSI C library standard. It is therefore not available if you use the compilers with the `-strict` option.

Syntax

```
lldiv_t lldiv(long long num, long long denom)
```

4.3.7 llabs()

The `llabs()` returns the absolute value of its input. It is the **`long long`** equivalent of the ANSI function `labs`.

`<stdlib.h>` is an ANSI header file, but the function is not allowed by the ANSI C library standard. It is therefore not available if you use the compilers with the `-strict` option.

Syntax

```
long long llabs(long long num)
```

4.3.8 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 8-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)
```

4.3.9 _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 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.

4.3.10 __heapstats()

The `__heapstats()` function displays statistics on the state of the storage allocation heap. It calls the `__Heap_Stats()` function, which you can re-implement if you choose to do your own storage management. The ARM default implementation gives information on how many free blocks exist, and estimates their size ranges.

Example 4-1 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 4-1 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`*, which 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, as sequence such as:

```
  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)
```

4.3.11 __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*, which 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 4-2 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 4-2 will cause a major failure 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)
```


4.4 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`	ANSI C and C++ basic runtime support
	`f`	C/Java rounding and exception options for fp arithmetic
	`g`	Full IEEE rounding and exception options for fp arithmetic
	`m`	Transcendental math functions
	`cpp`	High-level C++ functions that do not require fp arithmetic
	`cppfp`	High-level C++ functions that do require fp arithmetic.
arch	`a`	An ARM library
	`t`	A Thumb library (`-apcs interworking`).
fpu	`_p`	Soft VFP (`-fpu softvfp`)
	`__`	Does not use floating-point instructions (`-fpu none`).
stack	`u`	Does not use software stack checking (`-apcs /noswst`)
	`_`	Not applicable.
entrant	`n`	The functions are not reentrant (`-apcs /norwpi`)
	`e`	The functions are reentrant (`-apcs /rwpi`)
	`_`	Not applicable.
endian	`l`	Little-endian (`-li`)
	`b`	Big-endian (`-bi`).

The C library names are `c_{a,t}__un`

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

The standard FPLIB names are `f_{a,t}_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 standard IEEE names are `g_{a,t}_p`

g_a_p	ARM, soft VFP
g_t_p	Thumb, pure-endian double

The MATHLIB names are `m_{a,t}_pu`

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

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

See *Specifying the target processor or architecture* on page 2-19 for details on selecting a specific architecture or processor selection.

Chapter 5
Floating-point Support

This chapter describes the ARM support for floating-point computations. It contains the following sections:

5.1 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 5-30 for details of the ARM implementation of the standard.

An ARM system might have:

- a *Vector Floating-Point* (VFP) coprocessor
- no floating-point hardware.

RealView Compilation Tools for BREW compilers implement floating-point calculations in software. There is no support for VFP or FPA hardware implementations.

5.2 The software floating-point library, fplib

RealView Compilation Tools for BREW compilers compile programs to use a software floating-point. There is no floating-point instruction set available. RealView Compilation Tools for BREW 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 5-1 on page 5-4
- Table 5-2 on page 5-5
- Table 5-3 on page 5-6
- Table 5-4 on page 5-7.

User programs can call these routines directly. 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.

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.

5.2.1 Arithmetic on numbers in a particular format

The routines in Table 5-1 perform arithmetic on numbers in a particular format. Arguments and results are always in the same format.

Table 5-1 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 such 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 5-2 on page 5-5

5.2.2 Conversions between floats, doubles, and ints

The routines in Table 5-2 perform conversions between number formats, excluding **long long**s.

Table 5-2 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.

5.2.3 Conversions between long longs and other number formats

The routines in Table 5-3 perform conversions between `long long`s and other number formats.

Table 5-3 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.

5.2.4 Floating-point comparisons

The routines in Table 5-4 perform comparisons between floating-point numbers.

Table 5-4 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*).

5.3 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.

5.3.1 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. `__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 5-1).

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 5-1 IEEE status word layout

The fields in Figure 5-1 are as follows:

- Bits 0 to 4 are the sticky flags, exactly as described in *The __fp_status function* on page 5-10.
- Bits 8 to 12 are the exception mask bits, exactly as described in *The __fp_status function* on page 5-10, 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 5-5).

Table 5-5 Rounding mode control

Bits	Rounding mode
`00`	Round to nearest
`01`	Round up
`10`	Round down
`11`	Round toward zero

- 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);
```

5.3.2 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 5-6).

Table 5-6 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 5-2.

31 24	23 21	20 16	15 13	12 8	7 5	4 0
System ID	R	Masks	R	FPA only	R	Sticky

Figure 5-2 Floating-point status word layout

The fields in Figure 5-2 on page 5-10 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 5-35.
- 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);
```

5.3.3 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` allows 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 5-7.

Table 5-7 _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

Table 5-7 _controlfp argument macros (continued)

Macro	Description
`_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 5-7 on page 5-12, 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 5-7 on page 5-12, for example `_EM_INVALID`, `_EM_ZERODIVIDE`, can be used to test bits of the returned result.

`_statusfp` has the following prototype:

```
unsigned _statusfp(void);
```

5.3.4 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 allows 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 5-17.

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 (which operation was being performed 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.

5.3.5 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 5-8).
- 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:

```
__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;
      long long l;
      unsigned long long ul;
      struct { int word1, word2; } str;
  } __ieee_value_t;
  ```

- `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 5-8).
- `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 5-8.

Table 5-8 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 5-9.

Table 5-9 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 5-10.

Table 5-10 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 5-11.

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

Table 5-11 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 5-1 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 5-1.

Example 5-1

```
__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 ANSI function raise. In Example 5-1 on page 5-21, instead of:

```
    raise(SIGFPE);
```

it is better to code:

```
    __rt_raise(SIGFPE, _FPE_INVALID);
```

`__rt_raise` is declared in `rt_misc.h`.

5.4 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 ANSI C standard, mathlib provides the following functions:

- *Inverse hyperbolic functions (acosh, asinh, atanh)* on page 5-25
- *Cube root (cbrt)* on page 5-25
- *Copy sign (copysign)* on page 5-25
- *Error functions (erf, erfc)* on page 5-25
- *One less than exp(x) (expm1)* on page 5-26
- *Determine if a number is finite (finite)* on page 5-26
- *Gamma function (gamma, gamma_r)* on page 5-26
- *Hypotenuse function (hypot)* on page 5-26
- *Return the exponent of a number (ilogb)* on page 5-27
- *Determine if a number is a NaN (isnan)* on page 5-27
- *Bessel functions of the first kind (j0, j1, jn)* on page 5-27
- *The logarithm of the gamma function (lgamma, lgamma_r)* on page 5-27
- *Logarithm of one more than x (log1p)* on page 5-28
- *Return the exponent of a number (logb)* on page 5-28
- *Return the next representable number (nextafter)* on page 5-28
- *IEEE 754 remainder function (remainder)* on page 5-28
- *IEEE round-to-integer operation (rint)* on page 5-28
- *Scale a number by a power of two (scalb, scalbn)* on page 5-29
- *Return the fraction part of a number (significand)* on page 5-29
- *Bessel functions of the second kind (y0, y1, yn)* on page 5-29.

5.4.1 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 ANSI-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.

5.4.2 Cube root (cbrt)

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

This function returns the cube root of its argument.

5.4.3 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.

5.4.4 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.

5.4.5 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.

5.4.6 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.

5.4.7 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 5-27).

Note

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

5.4.8 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`**.

5.4.9 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.

5.4.10 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.

5.4.11 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. `jn` 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.

5.4.12 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.

5.4.13 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.

5.4.14 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.

5.4.15 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.

5.4.16 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 5-4).

5.4.17 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 5-4).

5.4.18 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.

5.4.19 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).

5.4.20 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.

5.5 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.

5.5.1 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 5-3.

31	30 23	22 0
S	Exp	Frac

Figure 5-3 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 5-4 shows its structure.

63	62 ... 52	51 ... 0
S	Exp	Frac

Figure 5-4 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 5-12 and Table 5-13 on page 5-33.

Table 5-12 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 5-35) when used, whereas quiet NaNs do not.

Table 5-13 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 5-12 on page 5-32.

5.5.2 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.

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.

5.5.3 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, which 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 5-12 on page 5-32) 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.

Appendix A
Via File Syntax

This appendix describes the syntax of via files accepted by RealView Compilation Tools for BREW compilers, linker, assembler, and fromELF. It contains the following sections:

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 the following tools:

- the compilers
- the assembler
- the linker.

You 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:

```
-split -ropi        -- Treated as two words
-split-ropi         -- Treated as one word
```

- The end of a line is treated as white space. For example:

```
-split
-ropi
```

  is equivalent to:

```
-split -ropi
```

- 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    -- Three words
-libpath "c:\Program Files\ARM"  -- Two words
```

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

```
-DNAME='"ARM RealView Compilation Tools for BREW"'      -- 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"
```

  is treated as the single word:

```
-IC:\Program Files\ARM
```

- Lines beginning with a semicolon (;) or a hash (#) character as the first non-whitespace character are comment lines. If a semicolon or hash character appears anywhere else in 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 RealView Compilation Tools for BREW C compiler and ANSI C library, not defined by the ISO C standard, that are implementation-defined:

- *Translation* on page B-3
- *Environment* on page B-3
- *Identifiers* on page B-5
- *Characters* on page B-5
- *Integers* on page B-5
- *Floating-point* on page B-5
- *Arrays and pointers* on page B-5
- *Registers* on page B-6
- *Structures, unions, enumerations, and bitfields* on page B-6
- *Qualifiers* on page B-7
- *Declarators* on page B-7
- *Statements* on page B-7
- *Preprocessing directives* on page B-7
- *Library functions* on page B-9.

Note

Nonconformance with ANSI

The compiler behavior differs from the behavior described in the language conformance sections of the C standard in that there is no support for the `wctype.h` and `wchar.h` headers.

B.1.1 Translation

Diagnostic messages produced by the compiler are of the form:

`source-file`, *`line-number`*: *`severity`*: *`error-code`*: *`explanation`*

where *`severity`* is one of:

`Warning`	This is a helpful message from the compiler relating to a minor violation of the ANSI specification.
`Error`	This is a violation of the ANSI specification but the compiler is able to recover by guessing the intention.
`Serious error`	This is a violation of the ANSI specification and no recovery is possible because the intention is not clear.
`Fatal error`	This is an indication that the compiler limits have been exceeded, or that the compiler has detected an internal fault (for example, not enough memory).

`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-based environment into arguments to `main()` is implementation-specific. The generic RealView Compilation Tools for BREW 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 will not be recognized inside double quotes.

Interactive device

In an unhosted implementation of the RealView Compilation Tools for BREW C library, the term *interactive device* might be meaningless. The generic RealView Compilation Tools for BREW 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 RealView Compilation Tools for BREW 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` to *`filename`*.

`2> filename`

This writes `stderr` to *`filename`*.

`2>&1` This writes `stderr` to the same place as `stdout`.

`>& file`

This writes both `stdout` and `stderr` to *`filename`*.

`>> filename`

This appends `stdout` to *`filename`*.

`>>& filename`

This appends both `stdout` and `stderr` to *`filename`*.

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 3-20 for details.

B.1.4 Characters

See *Character sets and identifiers* on page 3-20 for details.

B.1.5 Integers

See *Integer* on page 3-22 for details.

B.1.6 Floating-point

See *Float* on page 3-23 for details.

B.1.7 Arrays and pointers

See *Arrays and pointers* on page 3-23 for details.

B.1.8 Registers

Using RealView Compilation Tools for BREW compilers, you can declare any number of local objects (auto variables) to have the storage class `register`. See *Variable declaration keywords* on page 3-9 for information on how RealView Compilation Tools for BREW compilers implement 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 3-20.

Unions

See *Unions* on page 3-25 for details.

Enumerations

See *Enumerations* on page 3-25 for details.

Padding and alignment of structures

See *Structures* on page 3-25 for details.

Bitfields

See *Bitfields* on page 3-27 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 RealView Compilation Tools for BREW compilers.

B.1.13 Preprocessing directives

The ANSI standard C header files are stored within the compiler and 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 ANSI C library variants are listed in *About the runtime libraries* on page 4-2.

The precise nature of each C library is unique to the particular implementation. The generic RealView Compilation Tools for BREW 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 will be 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 4-2.

Appendix C
Standard C++ Implementation Definition

The majority of the language features described in the ISO/IEC standard for C++ are supported by RealView Compilation Tools for BREW C++ compilers. This appendix lists the C++ language features defined in the standard, and states whether or not that language feature is supported by RealView Compilation Tools for BREW C++.

——— **Note** ———

ARM C++ differs from ISO/IEC because the compliance requirements for Embedded C++ (EC++) differ from the requirements for ISO/IEC C++.

This section does not duplicate information that is part of the standard C implementation. See Appendix B *Standard C Implementation Definition* .

———————————

When used in ANSI C mode, the RealView Compilation Tools for BREW C++ compilers are identical to the RealView Compilation Tools for BREW C compilers. 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 3-16.

C.1 EC++ support

ARM C++ supports all features required by the definition of Embedded C++ except for argument-dependent name lookup (Koenig lookup).

C.2 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.3 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.

C.4 Minor features of language support

Table C-1 shows the minor features of the language supported by this release of ARM C++.

Table C-1 Minor feature support for language

Minor feature	Support
`atexit`	Implemented as defined in *The Annotated C++ Reference*, Addison-Wesley, 1991.
Namespaces	No.
Runtime type identification (RTTI)	Partial. `Typeid` is supported for static types and expressions with non-polymorphic type. See also the restrictions on new style casts.
New style casts	Partial. RealView Compilation Tools for BREW C++ supports the syntax of new style casts, but does not enforce the restrictions. New style casts behave in the same manner as old style casts.
Array new/delete	Yes.
Nothrow `new`	Yes
`bool` type	Yes.
`wchar_t` type	Partial, an implicit typedef for `unsigned short`.
`explicit` keyword	Yes.
Static member constants	Yes.
`extern inline`	Yes.
Full linkage specification	Yes.
`for` loop variable scope change	Yes.
Covariant return types	Yes (but not for non-leftmost base classes).
Default template arguments	Partial (args not dependent on other template args).
Template instantiation directive	Yes.

Table C-1 Minor feature support for language (continued)

Minor feature	Support
Template specialization directive	Yes.
`typename` keyword	Yes.
Member templates	Yes.
Partial specialization for class template	Yes.
Partial ordering of function templates	Yes.
Universal character names	No.
Koenig lookup	No.

C.5 Major features of language support

Table C-2 shows the major features of the language supported by this release of RealView Compilation Tools for BREW C++.

Table C-2 Major feature support for language

Major feature	ISO/IEC standard section	Support
Core language	1 to 13	Yes
Templates	14	Templates are partially supported
Exceptions	15	No
Libraries	17 to 27	See the *Standard C++ library implementation definition* on page C-8 and to Chapter 4 *The C and C++ Libraries*

C.6 Standard C++ library implementation definition

RealView Compilation Tools for BREW does not supply a standard C++ library.

Appendix D
C and C++ Compiler Implementation Limits

This appendix list the implementation limits for the RealView Compilation Tools for BREW C and C++ compilers. It contains the following sections:

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 RealView Compilation Tools for BREW compilers, 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 will be 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 RealView Compilation Tools for BREW 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 would be interpreted as this value by the RealView Compilation Tools for BREW compilers.

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 vii).

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 in RealView Compilation Tools for BREW compilers.

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

Glossary

American National Standards Institute (ANSI)
An organization that specifies standards for, among other things, computer software.

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-Thumb Procedure Call Standard (ATPCS)
ARM-Thumb Procedure Call Standard defines how registers and the stack will 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.

EC++ A variant of C++ designed to be used for embedded applications.

ELF Executable and linking format.

Environment The actual hardware and operating system that an application will run on.

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

Execution view The address of regions and sections after the image has been loaded into memory and started execution.

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.

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 on which its default thread runs.

Inline Functions that are repeated in code each time they are used rather than having a common subroutine.

See also Output sections.

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.

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.

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

PIC Position Independent Code.

See also ROPI.

PID Position Independent Data.

See also RWPI.

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.

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.

RWPI *See* Read Write Position Independent.

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.

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.

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

Veneer A small block of code used with subroutine calls when there is a requirement to change processor state or branch to an address that cannot be reached in the current processor state.

VFP *See* Vector Floating Point.

Volatile Memory addresses where the contents can change independently of the executing application are described as volatile. These are typically memory-mapped peripherals.

See also Memory mapped

Word A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

Zero Initialized (ZI) R/W memory used to hold variables that do not have an initial value. The memory is normally set to zero on reset.

ZI *See* Zero Initialized.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

D

E

F

M

N

O

P

Q

R

S

T

U

V

W

Symbols

