RealView® Compilation Tools

Version 4.0

Compiler User Guide

ARM®

ARM DUI 0205J (ID101213)

RealView Compilation Tools
Compiler User Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
August 2002	A	Non-Confidential	Release 1.2
January 2003	B	Non-Confidential	Release 2.0
September 2003	C	Non-Confidential	Release 2.0.1 for RealView Developer Suite v2.0
January 2004	D	Non-Confidential	Release 2.1 for RealView Developer Suite v2.1
December 2004	E	Non-Confidential	Release 2.2 for RealView Developer Suite v2.2
May 2005	F	Non-Confidential	Release 2.2 for RealView Developer Suite v2.2 SP1
March 2006	G	Non-Confidential	Release 3.0 for RealView Development Suite v3.0
March 2007	H	Non-Confidential	Release 3.1 for RealView Development Suite v3.1
September 2008	I	Non-Confidential	Release 4.0 for RealView Development Suite v4.0
January 2009	I	Non-Confidential	Update 1 for RealView Development Suite v4.0
15 April 2010	I	Non-Confidential	Update 2 for RealView Development Suite v4.0
10 December 2010	J	Non-Confidential	Update 3 for RealView Development Suite v4.0

Proprietary Notice

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

`http://www.arm.com`

Contents
RealView Compilation Tools Compiler User Guide

Preface

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

About this book

This book provides you with information on *RealView Compilation Tools* (RVCT), and gives an overview of the command-line options and compiler-specific features that are supported by the ARM compiler and the NEON™ vectorizing compiler.

Intended audience

This book is written for all developers who are producing applications using RVCT. It assumes that you are an experienced software developer. See the *RealView Compilation Tools Essentials Guide* for an overview of the ARM development tools provided with RVCT.

Using this book

This book is organized into the following chapters and appendixes:

Chapter 1 ***Introduction***

Read this chapter for an overview of the ARM compiler, the conformance standards and the C and C++ Libraries.

Chapter 2 ***Getting started with the ARM Compiler***

Read this chapter for an overview of the command-line options and compiler-specific features. It describes how to invoke the compiler, how to pass options to other RVCT tools and how to control diagnostic messages.

Chapter 3 ***Using the NEON Vectorizing Compiler***

Read this chapter for a tutorial on the NEON vectorizing compiler. It provides you with an understanding of the NEON unit and explains how to take advantage of the automatic vectorizing features.

Chapter 4 ***Compiler Features***

Read this chapter for an overview of the intrinsics supported by the ARM compiler.

Chapter 5 ***Coding Practices***

Read this chapter for an overview of good programming practice for RVCT.

Chapter 6 ***Diagnostic Messages***

Read this chapter for an overview of the diagnostic messages produced by the RVCT tools.

Chapter 7 ***Using the Inline and Embedded Assemblers***

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

This book assumes that the ARM software is installed in the default location. For example, on Windows this might be *volume*`:\Program Files\ARM`. This is assumed to be the location of *install_directory* when referring to path names, for example *install_directory*`\Documentation\...`. You might have to change this if you have installed your ARM software in a different location.

Typographical conventions

The following typographical conventions are used in this book:

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

`mono`space Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

`monospace italic`

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

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 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://infocenter.arm.com/help/index.jsp` for current errata sheets and addenda, and the ARM Frequently Asked Questions (FAQs).

ARM publications

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

- *RVCT Essentials Guide* (ARM DUI 0202)
- *RVCT Compiler Reference Guide* (ARM DUI 0348)
- *RVCT Libraries and Floating Point Support Guide* (ARM DUI 0349)
- *RVCT Linker User Guide* (ARM DUI 0206)
- *RVCT Linker Reference Guide* (ARM DUI 0381)
- *RVCT Utilities Guide* (ARM DUI 0382)
- *RVCT Assembler Guide* (ARM DUI 0204)
- *RVCT Developer Guide* (ARM DUI 0203)

A glossary is provided in the *RVDS Getting Started Guide*.

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

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

- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406)
- *ARMv7-M Architecture Reference Manual* (ARM DDI 0403)
- *ARMv6-M Architecture Reference Manual* (ARM DDI 0419)
- ARM datasheet or technical reference manual for your hardware device.

Other publications

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

The following publications describe the C++ language:

- *ISO/IEC 14882:2003, C++ Standard*.
- Stroustrup, B., *The C++ Programming Language* (3rd edition, 1997). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-88954-4.

The following books provide general C++ programming information:

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

  This book explains how C++ evolved from its first design to the language in use today.
- Vandevoorde, D and Josuttis, N.M. *C++ Templates: The Complete Guide* (2003). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-73484-2.
- Meyers, S., *Effective C++* (1992). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-56364-9.

  This provides short, specific, guidelines for effective C++ development.
- Meyers, S., *More Effective C++* (2nd edition, 1997). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-92488-9.

The following publications provide general C programming information:

- ISO/IEC 9899:1999, *C Standard.*

  The standard is available from national standards bodies (for example, AFNOR in France, ANSI in the USA).
- Kernighan, B.W. and Ritchie, D.M., *The C Programming Language* (2nd edition, 1988). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-110362-8.

  This book is co-authored by the original designer and implementor of the C language, and is updated to cover the essentials of ANSI C.
- Harbison, S.P. and Steele, G.L., *A C Reference Manual* (5th edition, 2002). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-089592-X.

  This is a very thorough reference guide to C, including useful information on ANSI C.
- Plauger, P., *The Standard C Library* (1991). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-131509-9.

  This is a comprehensive treatment of ANSI and ISO standards for the C Library.
- Koenig, A., *C Traps and Pitfalls,* Addison-Wesley (1989), Reading, Mass. ISBN 0-201-17928-8.

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

See `http://www.dwarfstd.org` for the latest information about the *Debug With Arbitrary Record Format* (DWARF) debug table standards and *Executable and Linking Format* (ELF) specifications.

The following publications provide information about the *European Telecommunications Standards Institute* (ETSI) basic operations. They are all available from the telecommunications bureau of the *International Telecommunications Union* (ITU) at `http://www.itu.int`.

- ETSI Recommendation G.191: *Software tools for speech and audio coding standardization*
- *ITU-T Software Tool Library 2005 User's manual*, included as part of ETSI Recommendation G.191
- ETSI Recommendation G723.1: *Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s*
- ETSI Recommendation G.729: *Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP).*

Publications providing information about Texas Instruments compiler intrinsics are available from Texas Instruments at `http://www.ti.com`.

Feedback

ARM welcomes feedback on both RVCT and the documentation.

Feedback on RealView Compilation Tools

If you have any problems with RVCT, contact your supplier. To help them provide a rapid and useful response, give:

- your name and company
- the serial number of the product
- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tools, including the version number and build numbers.

Feedback on this book

If you notice any errors or omissions in this book, send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the ARM® compiler provided with RVCT. It describes the standards of conformance and gives an overview of the runtime libraries provided with RVCT. It contains the following sections:

1.1 About the ARM compiler

The ARM compiler, `armcc`, is an optimizing C and C++ compiler that compiles Standard C and Standard C++ source code into machine code for ARM architecture-based processors. It complies with the *Base Standard Application Binary Interface for the ARM Architecture (*BSABI) and generates output objects in ELF format with support for DWARF 3 debug tables. It uses the *Edison Design Group* (EDG) front-end.

If you are upgrading to RVCT from a previous release or are new to RVCT, ensure that you read *RVCT Essentials Guide* for the most recent information.

1.2 About the NEON vectorizing compiler

NEON™ is an implementation of the ARM Advanced Single Instruction, Multiple Data (SIMD) Extension.

RVCT provides `armcc --vectorize`, a vectorizing mode of the ARM compiler, that targets ARM processors with a NEON unit, such as the Cortex-A8 and the Cortex-A9.

Note

To compile for Cortex-A9 targets, you must have a license for RealView Development Suite Professional.

Vectorizing means that the compiler generates NEON vector instructions directly from C or C++ code.

As an alternative to automatic compiler vectorization, RVCT also supports NEON intrinsics as an intermediate step for SIMD code generation between a vectorizing compiler and writing assembler code.

See:

- Chapter 3 *Using the NEON Vectorizing Compiler*
- *NEON Intrinsics* on page 4-9
- Appendix E *Using NEON Support* in the *Compiler Reference Guide*.

1.3 Source language modes

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

ISO C90 The ARM compiler compiles C as defined by the 1990 C standard and addenda.

Use the compiler option `--c90` to compile C90 code. This is the default.

ISO C99 The ARM compiler compiles C as defined by the 1999 C standard and addenda.

Use the compiler option `--c99` to compile C99 code.

ISO C++ The ARM compiler compiles C++ as defined by the 2003 standard, excepting wide streams and export templates.

Use the compiler option `--cpp` to compile C++ code.

The compiler provides support for numerous extensions to the C and C++ languages. For example, some GNU compiler extensions are supported. The compiler has several modes where compliance to a source language is either enforced or relaxed:

Strict mode In strict mode the compiler enforces compliance with the language standard relevant to the source language.

To compile in strict mode, use the command-line option `--strict`.

GNU mode In GNU mode all the GNU compiler extensions to the relevant source language are available.

To compile in GNU mode, use the compiler option `--gnu`.

For more information on source language modes and language compliance, see *New features of C99* on page 5-45. In addition, see:

- *Source language modes* on page 1-3 in the *Compiler Reference Guide*
- *Language extensions and language compliance* on page 1-5 in the *Compiler Reference Guide*
- *--c90* on page 2-22 in the *Compiler Reference Guide*
- *--c99* on page 2-22 in the *Compiler Reference Guide*
- *--cpp* on page 2-30 in the *Compiler Reference Guide*
- *--gnu* on page 2-67 in the *Compiler Reference Guide*
- *--strict, --no_strict* on page 2-119 in the *Compiler Reference Guide*.

1.4 The C and C++ libraries

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

The ARM C libraries

The ARM C libraries provide standard C functions, and helper functions used by the C and C++ libraries.

The ARM libraries comply with:

- the *C Library ABI for the ARM Architecture* (CLIBABI)
- the *C++ ABI for the ARM Architecture* (CPPABI).

See:

- *The C and C++ libraries* on page 1-8 in the *Compiler Reference Guide*
- *ABI for the ARM Architecture compliance* on page 1-4 in the *Libraries Guide*.

Rogue Wave Standard C++ Library version 2.02.03

The Rogue Wave Standard C++ Library, as supplied by Rogue Wave Software, Inc., provides standard C++ functions and objects such as `cout`. It also includes data structures and algorithms known as the *Standard Template Library* (STL).

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

Support libraries

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

The C and C++ libraries are provided as binaries only. There are variants of the C and C++ libraries for each combination of major build options, such as the byte order of the target system, whether interworking is selected, and whether floating-point support is selected.

See Chapter 2 *The C and C++ Libraries* in the *Libraries Guide*.

Chapter 2
Getting started with the ARM Compiler

This chapter outlines the command-line options accepted by the ARM compiler, `armcc`. It describes how to invoke the compiler, how to pass options to other RVCT tools and how to control diagnostic messages. It contains the following sections:

See the *Compiler Reference Guide*.

2.1 Using command-line options

You can control many aspects of compiler operation with command-line options.

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

Single-letter options

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

`-J directory`

`-Jdirectory`

Keyword options

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

`--depend=file.d`

`--depend file.d`

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

To compile files with names starting with a dash, use the POSIX option -- to specify that all subsequent arguments are treated as filenames, not as command switches. For example, to compile a file named `-ifile_1`, use:

```
armcc -c -- -ifile_1
```

2.1.1 Invoking the ARM compiler

The command for invoking the ARM compiler is:

```
armcc [help-options] [source-language] [search-paths] [project-template-options]
[PCH-options] [preprocessor-options] [C++-language] [output-format]
[target-options] [debug-options] [code-generation-options]
[optimization-options] [diagnostic-options] [additional-checks] [PCS-options]
[pass-thru-options] [arm-linux-options] [source]
```

See Chapter 2 *Compiler Command-line Options* in the *Compiler Reference Guide* for more information on each of the following options:

help-options Shows the main command-line options, the version number of the compiler and how the compiler has processed the command line:

- *--help* on page 2-71

- *--show_cmdline* on page 2-116
- *--vsn* on page 2-134.

source-language Specifies the source language variants accepted by the compiler:

- *--c90* on page 2-22
- *--c99* on page 2-22
- *--compile_all_input, --no_compile_all_input* on page 2-24
- *--cpp* on page 2-30
- *--gnu* on page 2-67
- *--strict, --no_strict* on page 2-119
- *--strict_warnings* on page 2-120.

These language options can be combined. For example:

```
armcc --c90 --gnu
```

search-paths Specifies the directories to search for included files:

- *-Idir[,dir,...]* on page 2-72
- *-Jdir[,dir,...]* on page 2-77
- *--kandr_include* on page 2-78
- *--preinclude=filename* on page 2-105
- *--reduce_paths, --no_reduce_paths* on page 2-109
- *--sys_include* on page 2-121.

See *Header files* on page 2-14 for more information on how these options work together.

project-template-options

Controls the behavior of project templates:

- *--project=filename, --no_project* on page 2-107
- *--reinitialize_workdir* on page 2-110
- *--workdir=directory* on page 2-136.

PCH-options Controls the processing of PCH files:

- *--create_pch=filename* on page 2-34
- *--pch* on page 2-101
- *--pch_dir=dir* on page 2-101
- *--pch_messages, --no_pch_messages* on page 2-102
- *--pch_verbose, --no_pch_verbose* on page 2-102
- *--use_pch=filename* on page 2-129.

preprocessor-options

Specifies preprocessor behavior, including preprocessor output and macro definitions:

- *-C* on page 2-22
- *--code_gen, --no_code_gen* on page 2-23
- *-Dname[(parm-list)][=def]* on page 2-35
- *-E* on page 2-52
- *-M* on page 2-88
- *-Uname* on page 2-127.

C++-language

Specifies options specific to C++ compilation:

- *--anachronisms, --no_anachronisms* on page 2-3
- *--dep_name, --no_dep_name* on page 2-39
- *--export_all_vtbl, --no_export_all_vtbl* on page 2-55
- *--force_new_nothrow, --no_force_new_nothrow* on page 2-57
- *--friend_injection, --no_friend_injection* on page 2-66
- *--guiding_decls, --no_guiding_decls* on page 2-70
- *--implicit_include, --no_implicit_include* on page 2-73
- *--implicit_include_searches, --no_implicit_include_searches* on page 2-73
- *--implicit_typename, --no_implicit_typename* on page 2-74
- *--nonstd_qualifier_deduction, --no_nonstd_qualifier_deduction* on page 2-94
- *--old_specializations, --no_old_specializations* on page 2-98
- *--parse_templates, --no_parse_templates* on page 2-100
- *--pending_instantiations=n* on page 2-103
- *--rtti, --no_rtti* on page 2-114
- *--using_std, --no_using_std* on page 2-130
- *--vfe, --no_vfe* on page 2-132.

output-format

Specifies the format for the compiler output. You can use these options to generate object files, assembly language output listing files, and make file dependency files:

- *--asm* on page 2-16
- *-c* on page 2-21
- *--default_extension=ext* on page 2-38
- *--depend=filename* on page 2-40

- *--depend_format=string* on page 2-41
- *--depend_system_headers, --no_depend_system_headers* on page 2-42
- *--info=totals* on page 2-74
- *--interleave* on page 2-76
- *--list* on page 2-82
- *--md* on page 2-89
- *-o filename* on page 2-95
- *-S* on page 2-114
- *--split_sections* on page 2-118.

target-options
Specifies the target processor or architecture and the target instruction set in use at startup:

- *--arm* on page 2-8
- *--compatible=name* on page 2-23
- *--cpu=list* on page 2-30
- *--cpu=name* on page 2-30
- *--fpu=list* on page 2-62
- *--fpu=name* on page 2-62
- *--thumb* on page 2-122.

See *Specifying the target processor or architecture* on page 2-23.

half-precision floating-point option
Enables the use of half-precision floating-point numbers as an optional extension to the VFPv3 architecture:

- *--fp16_format=format* on page 2-59.

debug-options
Controls the format and generation of debug tables:

- *--debug, --no_debug* on page 2-37
- *--debug_macros, --no_debug_macros* on page 2-37
- *--dwarf2* on page 2-51
- *--dwarf3* on page 2-51
- *-g* on page 2-66
- *--remove_unneeded_entities, --no_remove_unneeded_entities* on page 2-111.

code-generation-options

Specifies the code generation options for the ARM compiler, including endianness, symbol visibility, and alignment criteria:

- *--alternative_tokens, --no_alternative_tokens* on page 2-3
- *--bigend* on page 2-17
- *--bss_threshold=num* on page 2-20
- *--dllexport_all, --no_dllexport_all* on page 2-50
- *--dllimport_runtime, --no_dllimport_runtime* on page 2-50
- *--dollar, --no_dollar* on page 2-51
- *--enum_is_int* on page 2-53
- *--exceptions, --no_exceptions* on page 2-54
- *--exceptions_unwind, --no_exceptions_unwind* on page 2-54
- *--export_all_vtbl, --no_export_all_vtbl* on page 2-55
- *--export_defs_implicitly, --no_export_defs_implicitly* on page 2-55
- *--extended_initializers, --no_extended_initializers* on page 2-56
- *--hide_all, --no_hide_all* on page 2-71
- *--littleend* on page 2-85
- *--locale=lang_country* on page 2-86
- *--loose_implicit_cast* on page 2-87
- *--message_locale=lang_country[.codepage]* on page 2-90
- *--min_array_alignment=opt* on page 2-91
- *--multibyte_chars, --no_multibyte_chars* on page 2-92
- *--narrow_volatile_bitfields* on page 2-94
- *--pointer_alignment=num* on page 2-104
- *--restrict, --no_restrict* on page 2-112
- *--signed_bitfields, --unsigned_bitfields* on page 2-116
- *--signed_chars, --unsigned_chars* on page 2-117
- *--split_ldm* on page 2-117
- *--unaligned_access, --no_unaligned_access* on page 2-128
- *--vectorize, --no_vectorize* on page 2-131
- *--vla, --no_vla* on page 2-133
- *--wchar16* on page 2-135
- *--wchar32* on page 2-135.

optimization-options

Controls the level and type of code optimization:

- *--autoinline, --no_autoinline* on page 2-17
- *--data_reorder, --no_data_reorder* on page 2-36
- *--forceinline* on page 2-58
- *--fpmode=model* on page 2-59
- *--inline, --no_inline* on page 2-75
- *--library_interface=lib* on page 2-79
- *--library_type=lib* on page 2-81
- *--lower_ropi, --no_lower_ropi* on page 2-87
- *--lower_rwpi, --no_lower_rwpi* on page 2-87
- *--multifile, --no_multifile* on page 2-92
- *-Onum* on page 2-96
- *-Ospace* on page 2-99
- *-Otime* on page 2-99
- *--retain=option* on page 2-113.

Note

Optimization criteria can limit the debug information generated by the compiler.

diagnostic-options

Controls the diagnostic messages output by the compiler:

- *--brief_diagnostics, --no_brief_diagnostics* on page 2-19
- *--diag_error=tag[,tag,...]* on page 2-44
- *--diag_remark=tag[,tag,...]* on page 2-45
- *--diag_style={arm|ide|gnu}* on page 2-46
- *--diag_suppress=tag[,tag,...]* on page 2-47
- *--diag_suppress=optimizations* on page 2-47
- *--diag_warning=tag[,tag,...]* on page 2-48
- *--diag_warning=optimizations* on page 2-49
- *--errors=filename* on page 2-53
- *--remarks* on page 2-111
- *-W* on page 2-134
- *--wrap_diagnostics, --no_wrap_diagnostics* on page 2-137.

See Chapter 6 *Diagnostic Messages*.

command-line option file

Specifies a file containing additional command-line options:

- *--via=filename* on page 2-132.

multiple compilations

Specifies the feedback file that contains information about a previous build:

- *--feedback=filename* on page 2-56
- *--profile=filename* on page 2-107.

PCS-options Specifies the procedure call standard to use:

- *--apcs=qualifer...qualifier* on page 2-4.

See *Specifying the procedure call standard (AAPCS)* on page 2-24.

pass-thru-options Instructs the compiler to pass options to other RVCT tools:

- *-Aopt* on page 2-2
- *-Lopt* on page 2-79.

arm-linux-options Specifies options to configure RVCT for use with ARM Linux, and to build applications and shared libraries targeting ARM Linux:

- *--arm_linux_configure* on page 2-12
- *--arm_linux_config_file=path* on page 2-10
- *--configure_gcc=path* on page 2-27
- *--configure_gld=path* on page 2-28
- *--configure_sysroot=path* on page 2-29
- *--configure_cpp_headers=path* on page 2-24
- *--configure_extra_includes=paths* on page 2-25
- *--configure_extra_libraries=paths* on page 2-26
- *--arm_linux* on page 2-9
- *--arm_linux_paths* on page 2-13
- *--shared* on page 2-115
- *--translate_gcc* on page 2-124
- *--translate_g++* on page 2-122
- *--translate_gld* on page 2-125.

source Provides the filenames of one or more text files containing C or C++ source code. By default, the compiler looks for source files, and creates output files, in the current directory.

If a source file is an assembly file, that is, one with an extension of `.s`, the compiler activates the ARM assembler to process the source file.

This option is not used for *`arm-linux-options`*. It is mandatory for all other options.

The ARM compiler accepts one or more input files, for example:

```
armcc -c [options] ifile_1 ... ifile_n
```

Specifying a dash `-` for an input file causes the compiler to read from `stdin`. To specify that all subsequent arguments are treated as filenames, not as command switches, use the POSIX option `--`. See *Using command-line options* on page 2-2.

Default behavior

The compiler startup configuration is determined by the compiler according to the specified command-line options and the filename extensions. Command-line options override the default configuration determined by the filename extension. The compiler startup language can be C or C++ and the instruction set can be ARM or Thumb.

When you compile multiple files with a single command, all files must be of the same type, either C or C++. The compiler cannot switch the language based on the file extension. The following example produces an error, because the specified source files have different languages:

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

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

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

Where an unrecognized extension begins with `.c`, for example, *`filename`*`.cmd`, an error message is generated.

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

See *Precompiled header files* on page 2-17.

2.1.2 Ordering command-line options

In general, command-line options can appear in any order in a single compiler invocation. However, the effects of some options depend on the order they appear in the command line and how they are combined with other related options, for example, optimization options prefixed by -O, or PCH options. See *Precompiled header files* on page 2-17.

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

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

```
armcc -O1 -O2 -Ospace -Otime ...
```

is executed by the compiler as:

```
armcc -O2 -Otime
```

To see how the compiler has processed the command line, use the `--show_cmdline` option. This shows nondefault options that the compiler used. The contents of any via files are expanded. In the example used here, although the compiler executes `armcc -O2 -Otime`, the output from `--show_cmdline` does not include `-O2`. This is because `-O2` is the default optimization level, and `--show_cmdline` does not show options that apply by default.

2.1.3 Specifying command-line options with an environment variable

You can specify command-line options by setting the value of the `RVCT40_CCOPT` environment variable. The syntax is identical to the command line syntax. The compiler reads the value of `RVCT40_CCOPT` and inserts it at the front of the command string. This means that options specified in `RVCT40_CCOPT` can be overridden by arguments on the command-line.

2.1.4 Autocompleting command-line options

You can optionally request the autocompletion of command-line options. To do this, place a dot (`.`) after the characters to be autocompleted. Autocompletion only applies to keyword options.

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

You must include sufficient characters to make the autocompleted option unique. For example, use `--diag_su.=223` to specify `--diag_suppress=223` on the command line.

See *Using command-line options* on page 2-2.

2.1.5 Reading compiler options from a file

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

`--via` *`filename`*

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

See Appendix A *Via File Syntax* in the *Compiler Reference Guide*.

2.1.6 Specifying `stdin` input

Use minus (-) as the source filename to instruct the compiler to take input from `stdin`. The default compiler mode is C.

To terminate input, enter:

- `Ctrl-Z` then `Return` on Microsoft Windows systems
- `Ctrl-D` on Red Hat Linux systems.

An assembly listing for the keyboard input is sent to the output stream after input has been terminated if both the following are true:

- no output file is specified
- no preprocessor-only option is specified, for example `-E`.

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

2.2 File naming conventions

The ARM compiler uses filename suffixes to identify the classes of file involved in compilation and in the link stage. The filename suffixes recognized by the compiler are described in Table 2-1.

Table 2-1 Filename suffixes recognized by the ARM compiler

Suffix	Description	Usage notes
`.c`	C source file	Implies `--c90`

`.cpp`
`.c++`
`.cxx`
`.cc`
`.CC`
	C++ source file	Implies `--cpp`
		The compiler uses the suffixes `.cc` and `.CC` to identify files for implicit inclusion. See *Implicit inclusion* on page 5-15 in the *Compiler Reference Guide*.

`.d`	Dependency list file	`.d` is the default output filename suffix for files output using the `--md` option.
`.h`	C or C++ header file	`--cpp --arm`
`.o` `.obj`	ARM, Thumb, or mixed ARM and Thumb object file in ELF format.	
`.s`	ARM, Thumb, or mixed ARM and Thumb assembly language source file.	For files in the input file list suffixed with `.s`, the compiler invokes the assembler, `armasm`, to assemble the file. `.s` is the default output filename suffix for files output using either the option `-S` or `--asm`.
`.lst`	Error and warning list file	`.lst` is the default output filename suffix for files output using the `--list` option.
`.pch`	Precompiled header file	`.pch` is the default output filename suffix for files output using the `--pch` option.
`.txt`	Text file	`.txt` is the default output filename suffix for files output using the `-S` or `--asm` option in combination with the `--interleave` option.

2.2.1 Portability

To assist portability between hosts, use the following guidelines:

- Ensure that filenames do not contain spaces. If you have to use path names or filenames containing spaces, enclose the path and filename in double (") or single (') quotes.
- Make embedded path names relative rather than absolute.
- Use forward slashes (/) in embedded path names, not backslashes (\).

2.2.2 Output files

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

2.3 Header files

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

- the value of the environment variable `RVCT40INC`
- the `-I` and `-J` compiler options
- the `--kandr_include` and `--sys_include` compiler options
- whether the filename is an absolute filename or a relative filename
- whether the filename is between angle brackets or double quotes.

See:

- *-Idir[,dir,...]* on page 2-72 in the *Compiler Reference Guide*
- *-Jdir[,dir,...]* on page 2-77 in the *Compiler Reference Guide*
- *--kandr_include* on page 2-78 in the *Compiler Reference Guide*
- *--sys_include* on page 2-121 in the *Compiler Reference Guide*
- *Command-line options* on page 2-2 in the *Compiler Reference Guide*.

2.3.1 The current place

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

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

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

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

You can disable the stacking of current places by using the compiler option `--kandr_include`. This option makes the compiler use the search rule originally described by Kernighan and Ritchie in *The C Programming Language*. Under this rule each nonrooted user `#include` is searched for relative to the directory containing the source file that is being compiled. See *--kandr_include* on page 2-78 in the *Compiler Reference Guide*.

2.3.2 The RVCT40INC environment variable

The RVCT40INC environment variable points to the location of the included header and source files provided with RVCT. Do not change this environment variable. If you want to include files from other locations, use the -I and -J command-line options as required.

When compiling, directories specified with RVCT40INC are searched immediately after directories specified by the -I option have been searched. If you use the -J option, RVCT40INC is ignored.

2.3.3 The search path

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

Table 2-2 Include file search paths

Compiler option	<include> search order	"include" search order
Neither -I nor -J	RVCT40INC*dirs*	*CP*, RVCT40INC*dirs*
-I	RVCT40INC*dirs*, I*dirs*	*CP*, I*dirs*, RVCT40INC*dirs*
-J	J*dirs*	*CP*, and J*dirs*
Both -I and -J	J*dirs*, I*dirs*	*CP*, I*dirs*, J*dirs*
--sys_include	No effect	Removes *CP* from the search path
--kandr_include	No effect	Uses Kernighan and Ritchie search rules

In Table 2-2:

RVCT40INC*dirs*
: List of directories specified by the RVCT40INC environment variable, if set.

CP
: The current place.

I*dirs* and J*dirs*
: Directories specified by the -I*dirs* and -J*dirs* compiler options.

2.3.4 The TMP and TMPDIR environment variables

On Windows platforms, the environment variable `TMP` is used to specify the directory to be used for temporary files. If `TMP` is not defined, or if it is set to the name of a directory that does not exist, temporary files are created in the current working directory.

On Red Hat Linux platforms, the environment variable `TMPDIR` is used to specify the directory to be used for temporary files. If `TMPDIR` is not set, a default temporary directory, usually `/tmp` or `/var/tmp`, is used.

`TMP` and `TMPDIR` are typically set up by a system administrator. However, it is permissible for you to change them.

2.4 Precompiled header files

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

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

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

Note

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

Note

Do not assume that if a PCH file is available, it is used by the compiler. In some cases, system configuration issues (for example, Address Space Randomisation on RHE3 and Vista) mean that the compiler might not always be able to use the PCH file.

The ARM compiler can precompile header files automatically, or enable you to control the precompilation. See:

- *Automatic PCH processing*
- *Manual PCH processing* on page 2-20
- *Controlling the output of messages during PCH processing* on page 2-21
- *Performance issues* on page 2-21.

2.4.1 Automatic PCH processing

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

When the compiler creates a PCH file, it takes the name of the primary source file and replaces the suffix with `.pch`. The PCH file is created in the directory of the primary source file, unless you specify the `--pch_dir` option.

See *Ordering command-line options* on page 2-10.

The header stop point

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

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

Note

You can manually specify the header stop point with `#pragma hdrstop`. You must place this before the first token that does not belong to a preprocessing directive. In this example, place it before `int`. See *Controlling PCH processing* on page 2-21.

Conditions that affect PCH file generation

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

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

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

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

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

  ```
  // xxx.h
  class A
  {
      // xxx.c
  ```

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

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

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

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

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

  Note

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

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

- No instances, the `#line` preprocessing directive must appear.

- `#pragma no_pch` must not appear.

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

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

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

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

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

- if the PCH file is based on at least one out-of-date header file but is otherwise applicable for the current compilation
- if the PCH file has the same base name as the source file being compiled, for example, `xxx.pch` and `xxx.c`, but is not applicable for the current compilation, for example, because you have used different command-line options.

These describe some common cases. You must delete other PCH files as required.

2.4.2 Manual PCH processing

You can specify the filename and location of PCH files, and the parts of a header file that are subject to PCH processing.

Specifying a PCH filename and location

You can specify the filename and location of the PCH file using the following command-line options:

- `--create_pch=`*`filename`*
- `--use_pch=`*`filename`*
- `--pch_dir=`*`directory`*

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

Ordering PCH command-line options

The compiler cannot use these three options together on the same command line. If more than one of these options is specified, the following rule applies:

- `--use_pch` takes precedence over `--pch`
- `--create_pch` takes precedence over all other PCH options.

Most of the features of automatic PCH processing apply to one or other of these modes. For example, header stop points and PCH file applicability are determined in the same way.

Controlling PCH processing

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

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

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

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

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

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

Note

You can use these pragmas even if you are using automatic PCH processing.

See *Pragmas* on page 4-14.

2.4.3 Controlling the output of messages during PCH processing

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

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

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

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

2.4.4 Performance issues

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

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

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

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

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

2.5 Specifying the target processor or architecture

RVCT includes support for all ARM architectures from ARMv4 onwards, including ARM NEON™ technology. All architecture names before ARMv4 are now obsolete and no longer supported.

Specifying a target processor or architecture enables the compiler to take advantage of extra features specific to the selected processor or architecture. Use the `--cpu` and `--fpu` options to enable these features.

You can also specify the startup instruction set by using the `--arm` or `--thumb` option.

See:

- *NEON technology*
- *Selecting the target CPU* on page 5-3
- Chapter 5 *Interworking ARM and Thumb* in the *Developer Guide*.

See also, in the *Compiler Reference Guide*:

- *--arm* on page 2-8
- *--cpu=list* on page 2-30
- *--cpu=name* on page 2-30
- *--fpu=list* on page 2-62
- *--fpu=name* on page 2-62
- *--thumb* on page 2-122.

2.5.1 NEON technology

The ARM Advanced Single Instruction Multiple Data (SIMD) Extension, also known as NEON technology, is a 64/128-bit hybrid SIMD architecture developed by ARM to accelerate the performance of multimedia and signal processing applications. NEON is implemented as part of the processor, but has its own execution pipelines and a register bank that is distinct from the ARM register bank. Key features include aligned and unaligned data access, support for integer, fixed-point and single-precision floating point data types, tight coupling to the ARM core, and a large register file with multiple views. NEON instructions are available in both ARM and Thumb-2.

The ARM compiler provides support for Cortex™ processors equipped with a NEON unit. To generate NEON instructions you must specify a Cortex processor that includes NEON technology on the command line, for example, `--cpu=Cortex-A8`. There is no NEON support for architectures before ARMv7.

See Appendix E *Using NEON Support* in the *Compiler Reference Guide*.

2.6 Specifying the procedure call standard (AAPCS)

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

See:

- *Interworking qualifiers*
- *Position independence qualifiers*
- *Specifying the target processor or architecture* on page 2-23
- *Procedure Call Standard for the ARM Architecture* specification, `aapcs.pdf`, in *`install_directory`*`\Documentation\Specifications\....`

2.6.1 Interworking qualifiers

These `--apcs` qualifiers control interworking.

See:

- *--apcs=qualifer...qualifier* on page 2-4 in the *Compiler Reference Guide*
- Chapter 5 *Interworking ARM and Thumb* in the *Developer Guide*
- Chapter 3 *Using the Basic Linker Functionality* in the *Linker User Guide*.

2.6.2 Position independence qualifiers

These `--apcs` qualifiers control position independence. They also affect the creation of reentrant and thread-safe code.

See:

- *--apcs=qualifer...qualifier* on page 2-4 in the *Compiler Reference Guide*
- *Restrictions on position independent code and data* on page 2-25
- *Writing reentrant and thread-safe code* on page 2-4 in the *Libraries Guide*
- Chapter 4 *BPABI and SysV Shared Libraries and Executables* in the *Linker Reference Guide*.

Restrictions on position independent code and data

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

- The use of `--apcs /ropi` is not supported when compiling C++. You can compile only the C subset of C++ with `/ropi`.
- Some constructs that are legal C do not work when compiled for `--apcs=/ropi` or `--apcs=/rwpi`, for example:

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

  However, to enable these static initializations to work, use the `--lower_rwpi` and `--lower_ropi` options.

  To compile this code, type:

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

  You do not have to specify `--lower_rwpi`, because this is the default.
- The use of `--apcs=/fpic` is supported when compiling C++. Here, virtual table functions and `typeinfo` are placed in read-write areas so that they can be accessed relative to the location of the PC.
- If you use `--apcs=/fpic` , the compiler exports only functions and data marked `__declspec(dllexport)`.
- If you use `--no_hide_all`, the compiler uses STV_DEFAULT visibility for all `extern` variables and functions if they do not use `__declspec(dll*)`. The compiler disables auto-inlining of functions with STV_DEFAULT visibility.

  For example, use `--no_hide_all` and `--apcs /fpic` together when building a System V or ARM Linux shared library.

See *__declspec attributes* on page 4-24 in the *Compiler Reference Guide* for more information on the `__declspec` keyword.

See *Symbol visibility* on page 4-5 in the *Linker Reference Guide* for more information on symbol visibility.

2.7 Using linker feedback

The linker provides a feedback feature that enables:

- efficient elimination of unused functions
- reduction of compilation required for interworking.

2.7.1 Eliminating unused functions

Unused function code might occur in the following situations:

- Where you have legacy functions that are no longer used in your source code. Rather than manually remove the unused function code from your source code, you can use linker feedback to remove the unused object code automatically from the final image.
- When a function is inlined. If an inlined function is not declared as **`static`**, the out-of-line function code is still present in the object file, but there is no longer a call to that code.

To eliminate unused functions from your object files:

1. Compile your source code.
2. Use the linker option `--feedback=`*`filename`* to create a feedback file. By default, the type of feedback generated is for the elimination of unused functions.
3. Use the compiler option `--feedback=`*`filename`* to feed the feedback file to the compiler.

The compiler uses the feedback file generated by the linker to compile the source code in a way that enables the linker to subsequently discard the unused functions.

Note

To obtain maximum benefit from linker feedback, do a full compile and link at least twice. A single compile and link using feedback from a previous build is normally sufficient to obtain some benefit.

You can specify the `--feedback=`*`filename`* option even when no feedback file exists. This enables you to use the same build or make file regardless of whether a feedback file exists, for example:

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

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

See:

- *--feedback=filename* on page 2-56 in the *Compiler Reference Guide*
- *--feedback_type=type* on page 2-39 in the *Linker Reference Guide*
- *Feedback* on page 3-17 in the *Linker User Guide*.

2.7.2 Reduction of compilation required for interworking

Note

Reduction of compilation required for interworking is only applicable to ARMv4T architectures. ARMv5T and later processors can interwork without penalty.

The linker detects when an ARM function is being called from a Thumb state, and when a Thumb function is being called from an ARM state. You can use feedback from the linker to avoid compiling functions for interworking that are never used in an interworking context.

To reduce compilation required for interworking:

1. Compile your source code.
2. Use the linker options `--feedback=`*`filename`* and `--feedback_type=iw` to create a feedback file that reports functions requiring interworking support.
3. Use the compiler option `--feedback=`*`filename`* to feed the feedback file to the compiler.

The compiler uses the feedback file generated by the linker to compile the source code in a way that enables the compiler to subsequently avoid compiling functions for interworking if those functions are not used in an interworking context.

Note

Always ensure that you perform a full clean build immediately prior to using the linker feedback file. This minimizes the risk of the feedback file becoming out of date with the source code it was generated from.

See:

- *--feedback=filename* on page 2-56 in the *Compiler Reference Guide*

- *--feedback_type=type* on page 2-39 in the *Linker Reference Guide*
- *Feedback* on page 3-17 in the *Linker User Guide*.

2.8 Adding symbol versions

The compiler and linker support the GNU-extended symbol versioning model.

To create a function with a symbol version in C or C++ code, you must use the assembler label GNU extension to rename the function symbol into a symbol that has the name *function*`@@`*ver* for a default *ver* of *function*, or *function*`@`*ver* for a non default *ver* of *function*.

For example, to define a default version:

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

To define a non default version:

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

See:

- *Assembler labels* on page 3-20 in the *Compiler Reference Guide*
- *Symbol versioning* on page 4-19 in the *Linker User Guide*.

Chapter 3
Using the NEON Vectorizing Compiler

This chapter provides you with an understanding of the NEON™ unit and explains how to take advantage of the automatic vectorizing features. It contains the following sections:

3.1 The NEON unit

The NEON unit provides 32 vector registers that each hold 16 bytes of information. These 16 byte registers can then be operated on in parallel in the NEON unit. For example, in one vector add instruction you can add eight 16-bit integers to eight other 16 bit integers to produce eight 16-bit results.

The NEON unit supports 8-bit, 16-bit and 32-bit integer operations, and some 64-bit operations, in addition to 32-bit floating point operations.

Note

Vectorization of floating-point code does not always occur automatically. For example, loops that require reassociation only vectorize when compiled with `--fpmode fast`. Compiling with `--fpmode fast` enables the compiler to perform some transformations that could affect the result. (See *--fpmode=model* on page 2-59 in the *Compiler Reference Guide*.)

The NEON unit is classified as a vector SIMD unit that operates on multiple elements, in a vector register, with one instruction.

For example, array A is a 16-bit integer array with 8 elements.

Table 3-1 Array A

1	2	3	4	5	6	7	8

Array B has these 8 elements:

Table 3-2 Array B

80	70	60	50	40	30	20	10

To add these arrays together, fetch each vector into a vector register and use one vector SIMD instruction to obtain the result.

Table 3-3 Result

81	72	63	54	45	36	27	18

3.2 Writing code for NEON

This section gives an overview of the ways you can write code for the NEON unit. There are several ways to get code running on NEON:

- Write in assembly language, or use embedded assembly language in C, and use the NEON instructions directly.
- Write in C or C++ with the NEON C language extensions.
- Call a library routine that has been optimized to use NEON instructions.
- Use automatic vectorization to get loops vectorized for NEON.

3.2.1 NEON C extensions

The NEON C extensions are a set of new data types and intrinsic functions defined by ARM to enable access to the NEON unit from C. Most of the vector functions map directly to vector instructions available in the NEON unit and are compiled inline by the NEON enhanced ARM C compiler. With these extensions, performance can be achieved at the C level that is comparable to that obtained with assembly language coding.

See Appendix E *Using NEON Support* in the *Compiler Reference Guide*.

3.2.2 Automatic vectorization

By coding in vectorizable loops instead of writing in explicit NEON instructions, code portability is preserved between processors. Performance levels similar to that of hand coded vectorization are achieved with less effort.

Example 3-1 provides an example of command-line options to invoke automatic vectorization on the Cortex-A8 processor.

Example 3-1 Automatic vectorization

```
armcc --vectorize --cpu=Cortex-A8 -O3 -Otime -c file.c
```

Note

You can also compile with `-O2 -Otime`, however, this does not give the maximum code performance.

3.2.3 Performance goals

Most applications require tuning to gain the best performance from vectorization. There is always some overhead so the theoretical maximum performance cannot be reached. For example, the NEON unit can process four single-precision floats at one time. This means that the theoretical maximum performance for a floating-point application is a factor of four over the original scalar nonvectorized code. Given typical overheads, a reasonable goal for a whole floating-point application is to aim for a 50% improvement on performance over the scalar code. For large applications that are not completely vectorizable, achieving a 25% improvement on performance over the scalar code is a reasonable goal, although this is highly application dependent..

See *Improving performance* on page 3-7.

3.3 Working with automatic vectorization

This section gives an overview of automatic vectorization and also describes the factors affecting the vectorization process and performance of the generated code.

3.3.1 Overview of automatic vectorization

Automatic vectorization involves the high-level analysis of loops in your code. This is the most efficient way to map the majority of typical code onto the functionality of the NEON unit. For most code, the gains that can be made with algorithm-dependent parallelism on a smaller scale are very small relative to the cost of automatic analysis of such opportunities. For this reason, the NEON unit is designed as a target for loop-based parallelism.

Vectorization is carried out in a way that ensures that the optimized code gives the same results as the non vectorized code. In certain cases vectorization of a loop is not carried out so that the possibility of an incorrect result is avoided. This can lead to sub-optimal code, and you might have to manually tune your code to make it more suitable for automatic vectorization. See *Improving performance* on page 3-7.

3.3.2 Vectorization concepts

This section describes some concepts that are commonly used when considering vectorization of code.

Data references

Data references in your code can be classified as one of three types:

Scalar A single location that does not change through all the iterations of the loop.

Index An integer quantity that increments by a constant amount each pass through the loop.

Vector A range of memory locations with a constant stride between consecutive elements.

Example 3-2 on page 3-6 shows the classification of variables in the loop:

`i,j` index variables

`a,b` vectors

`x` scalar

Example 3-2 Categorization of a vectorizable loop

```
float *a, *b;
int i, j, n;
...
for (i = 0; i < n; i++)
{
    *(a+j) = x +b[i];
    j += 2;
};
```

To vectorize, the compiler has to identify variables with a vector access pattern. It also has to ensure that there are no data dependencies between different iterations of the loop.

Stride patterns and data accesses

The stride pattern of data accesses in a loop is the pattern of accesses to data elements between sequential loop iterations. For example, a loop that linearly accesses each element of an array has a stride of one. Another example, a loop that accesses an array with a constant offset between each element used, is described as having a constant stride. In Example 3-2, `b` is accessed with a stride of `1`, and `a` is accessed with a stride of 2.

3.3.3 Factors affecting vectorization performance

The automatic vectorization process and performance of the generated code is affected by the following:

The way loops are organized

For best performance, the innermost loop in a loop nest must access arrays with a stride of one.

The way the data is structured

The data type dictates how many data elements can be held in a NEON register, and therefore, how many operations can be performed in parallel.

The iteration counts of loops

Longer iteration counts are generally better, because the loop overhead is amortized over more iterations. Tiny iteration counts, such as two or three elements, can be faster to process with non vector instructions.

The data type of arrays

For example, NEON does not improve performance when double precision floating point arrays are used.

The use of memory hierarchy

Most current processors are relatively unbalanced between memory bandwidth and processor capacity. For example, performing relatively few arithmetic operations on large data sets retrieved from main memory is limited by the memory bandwidth of the system.

3.3.4 Improving performance

Most applications require some tuning on the part of the programmer to get the best NEON results. This section describes the different types of loops. It explains how vectorization works successfully with some loops but does not work with others. It also explains how you can modify code to achieve the best performance from the vectorized code.

General performance issues

Using the command-line options `-O3` and `-Otime` ensures that the code achieves significant performance benefits in addition to those of vectorization.

When optimizing for performance, you must give consideration to the high-level algorithm structure, data element size, array configurations, strict iterative loops, reduction operations and data dependency issues. Optimizing for performance requires an understanding of where in the program most of the time is spent. To gain maximum performance benefits you might have to use profiling and benchmarking of the code under realistic conditions.

Automatic vectorization can often be impeded by any earlier manual optimization of the code, for example, manual loop unrolling in the source code or complex array accesses. For optimal results, the best way is to write the code using simple loops, therefore enabling the compiler to perform all the optimization. For hand-optimized legacy code, you might find it easier to rewrite critical portions based on the original algorithm using simple loops.

See the following in the *Compiler Reference Guide*:

- *--vectorize, --no_vectorize* on page 2-131 in the *Compiler Reference Guide*
- *-Onum* on page 2-96 in the *Compiler Reference Guide*
- *-Otime* on page 2-99 in the *Compiler Reference Guide*.

Data dependencies

A loop that has results from one iteration feeding back into a future iteration of the same loop is said to have a data dependency conflict. The conflicting values might be array elements or a scalar such as an accumulated sum.

Loops containing data dependency conflicts might not be completely optimized. To detect data dependencies involving arrays and/or pointers requires extensive analysis of the arrays used in each loop nest, and examination of the offset and stride of accesses to elements along each dimension of arrays that are both used and stored in a loop. If there is a possibility of the usage and storage of arrays overlapping on different iterations of a loop, then there is a data dependency problem. A loop cannot be safely vectorized if the vector order of operations can change the results. In these cases, the compiler detects the problem and leaves the loop in its original form or carries out a partial vectorization of the loop. This type of data dependency must be avoided in your code to achieve the best performance.

In the loop shown in the Example 3-3, the reference to `a[i-2]` at the top of the loop conflicts with the store into `a[i]` at the bottom. Performing vectorization on this loop gives a different result, and so it is left in its original form.

Example 3-3 Non vectorizable data dependency

```
float a[99], b[99], t;
int i;
for (i = 3; i < 99; i++)
{
    t = a[i-1] + a[i-2];
    b[i] = t + 3.0 + a[i];
    a[i] = sqrt(b[i]) - 5.0;
};
```

Information from other array subscripts is used as part of the analysis of dependencies. The loop in Example 3-4 on page 3-9 vectorizes because the non vector subscripts of the references to array `a` can never be equal, because `n` is not equal to `n+1`, and so gives no feedback between iterations. The references to array `a` use two different pieces of the array and so, do not share data.

Example 3-4 Vectorizable data dependency

```
float a[99][99], b[99], c[99];
int i, n, m;
...
for (i = 1; i < m; i++) a[n][i] = a[n+1][i-1] * b[i] + c[i];
```

Scalar variables

A scalar variable that is used but not set, in a NEON loop is replicated in each position in a vector register and the result used in the vector calculation.

A scalar that is set and then used in a loop is *promoted* to a vector. These variables generally hold temporary scalar values in a loop that now has to hold temporary vector values. In Example 3-5, x is a *used* scalar and y is a *promoted* scalar.

Example 3-5 Vectorizable loop

```
float a[99], b[99], x, y;
int i, n;
...
for (i = 0; i < n; i++)
{
    y = x + b[i];
    a[i] = y + 1/y;
};
```

A scalar that is used and then set in a loop is called *a carry-around* scalar. These variables are a problem for vectorization because the value computed in one pass of the loop is carried forward into the next pass. In Example 3-6 x is a carry-around scalar.

Example 3-6 Non vectorizable loop

```
float a[99], b[99], x;
int i, n;
...
for (i = 0; i < n; i++)
{
    a[i] = x + b[i];
    x = a[i] + 1/x;
};
```

Reduction operations

A special category of scalar usages in a loop is reduction operations. This category involves the reduction of a vector of values down to a scalar result. The most common reduction is the summation of all elements of a vector. Other reductions include: dot product of two vectors, maximum value in a vector, minimum value in a vector, product of all vector elements and location of a maximum or minimum element in a vector.

Example 3-7 shows a dot product reduction where `x` is a reduction scalar.

Example 3-7 Dot product reduction

```
float a[99], b[99], x;
int i, n;
...
for (i = 0; i < n; i++) x += a[i] * b[i];
```

Reduction operations are worth vectorizing because they occur so often. In general, reduction operations are vectorized by creating a vector of partial reductions that are then reduced into the final resulting scalar.

Using pointers

When accessing arrays, the compiler can often prove that memory accesses do not overlap. When using pointers, this is less likely to be possible, and either requires a runtime test, or requires you to use **`restrict`**.

The compiler is able to vectorize loops containing pointers if it can determine that the loop is safe. Both array references and pointer references in loops are analyzed to see if there is any vector access to memory. In some cases, the compiler creates a run-time test, and executes a vector version or scalar version of the loop depending on the result of the test.

Often, function arguments are passed as pointers. If several pointer variables are passed to a function, it is possible that pointing to overlapping sections of memory can occur. Often, at runtime, this is not the case but the compiler always follows the safe method and avoids optimizing loops that involve pointers appearing on both the left and right sides of an assignment operator. For example, consider the function in Example 3-8 on page 3-11.

Example 3-8 Conditional vectorization of pointers

```
void func (int *pa, int *pb, int x)
{
    int i;
    for (i = 0; i < 100; i++) *(pa + i) = *(pb + i) + x;
};
```

In this example, if `pa` and `pb` overlap in memory in a way that causes results from one loop pass to feed back to a subsequent loop pass, then vectorization of the loop can give incorrect results. If the function is called with the following arguments, vectorization might be ambiguous:

```
int *a;

func (a, a-1);
```

The compiler performs a runtime test to see if pointer aliasing occurs. If pointer aliasing does not occur, it executes a vectorized version of the code. If pointer aliasing occurs, the original non vectorized code executes instead. This leads to a small cost in runtime efficiency and code size.

In practice, it is very rare for data dependence to exist because of function arguments. Programs that pass overlapping pointers are very hard to understand and debug, apart from any vectorization concerns.

See *restrict* on page 3-8 in the *Compiler Reference Guide*. In Example 3-8, adding **`restrict`** to `pa` is sufficient to avoid the runtime test.

Indirect addressing

Indirect addressing occurs when an array is accessed by a vector of values. If the array is being fetched from memory, the operation is called a *gather*. If the array is being stored into memory, the operation is called a *scatter*. In Example 3-9, `a` is being scattered, and `b` is being gathered.

Example 3-9 Non vectorizable indirect addressing

```
float a[99], b[99];
int ia[99], ib[99], i, n, j;
...
for (i = 0; i < n; i++) a[ia[i]] = b[j + ib[i]];
```

Indirect addressing is not vectorizable with the NEON unit because it can only deal with vectors that are stored consecutively in memory. If there is indirect addressing and significant calculations in a loop, it might be more efficient for you to move the indirect addressing into a separate non vector loop. This enables the calculations to vectorize efficiently.

Loop structure

The overall structure of a loop is important for obtaining the best performance from vectorization. Generally, it is best to write simple loops with iteration counts that are fixed at the start of the loop, and do not contain complex conditional statements or conditional exits. You might have to rewrite your loops to improve the vectorization performance of the code.

Exits from loops

Example 3-10 is also unable to vectorize because it contains an exit from the loop. In cases like this, you must rewrite the loop if possible for vectorization to succeed.

Example 3-10 Non vectorizable loop

```
int a[99], b[99], c[99], i, n;
...
for (i = 0; i < n; i++)
{
    a[i] = b[i] + c[i];
    if (a[i] > 5) break;
};
```

Loop iteration count

Loops must have a fixed iteration count at the start of the loop. Example 3-11 shows the iteration count is `n` and this is not changed through the course of the loop.

Example 3-11 Vectorizable loop

```
int a[99], b[99], c[99],i, n;
...
for (i = 0; i < n; i++) a[i] = b[i] + c[i];
```

Example 3-12 on page 3-13 has no fixed iteration count and is unable to vectorize automatically.

Example 3-12 Non vectorizable loop

```
int a[99], b[99], c[99], i, n;
...
while (i < n)
{
    a[i] = b[i] + c[i];
    i += a[i];
};
```

The NEON unit can operate on elements in groups of 2, 4, 8, or 16. Where the iteration count at the start of the loop is known, the compiler might add a runtime test to check if the iteration count is not a multiple of the lanes that can be used for the appropriate data type in a NEON register. This increases the code size because additional non vectorized code is generated to execute any additional loop iterations.

If you know that your iteration count is one of those supported by NEON, you can indicate this to the compiler. The most efficient way to do this is to divide the number of iterations by four in the caller and multiply by four in the function that you intend to vectorize. If you cannot modify all of the calling functions, you can use an appropriate expression for your loop limit test to indicate that the loop iteration is a suitable multiple. For example, to indicate that your loop is a multiple of four iterations, use:

```
for(i = 0; i < (n >> 2 << 2); i++)
```

or:

```
for(i = 0; i < (n & ~3); i++)
```

This reduces the size of the generated code and can give a performance improvement.

Writing loops to use all parts of a structure together is important for vectorization. Each part of the structure needs to be accessed within the same loop.

Example 3-13 Non vectorizable loops

```
for (...) { buffer[i].a = ....; }
for (...) { buffer[i].b = ....; }
for (...) { buffer[i].c = ....; }
```

Example 3-14 Vectorizable loop

```
for (...)
{
    buffer[i].a = ....;
    buffer[i].b = ....;
    buffer[i].c = ....;
}
```

Function calls and inlining

Calls to non-inline functions within a loop inhibit vectorization.

Splitting complex operations into several functions to aid clarity is common practice. In order for these functions to be considered for vectorization, they must be marked with the `__inline` or `__forceinline` keywords. These functions are then expanded inline for vectorization. See *__inline* on page 4-9 and *__forceinline* on page 4-6 in the *Compiler Reference Guide*.

Conditional statements

For efficient vectorization, loops must contain mostly assignments statements and limit the use of **`if`** and **`switch`** statements.

Simple conditions that do not change between iterations of the loop are described as being loop invariant. These can be moved before the loop by the compiler so that they do not have to be executed on each loop iteration. More complex conditional operations are vectorized by computing all pathways in vector mode and merging the results. If there is significant computation being performed conditionally, then a substantial amount of time is wasted.

Example 3-15 shows an acceptable use of conditional statements.

Example 3-15 Vectorizable condition

```
float a[99], b[99], c[i];
int i, n;
...
for (i = 0; i < n; i++)
{
    if (c[i] > 0) a[i] = b[i] - 5.0;
    else a[i] = b[i] * 2.0;
};
```

Structures

NEON structure loads require that all members of a structure are of the same length. Therefore, the compiler does not attempt to use vector loads for the code shown in Example 3-16.

Example 3-16 Non vectorizable code caused by inconsistent data types

```
struct foo
{
    short a;
    int b;
    short c;
} n[10];
```

The code in Example 3-16 could be rewritten for vectorization by using the same data type throughout the structure. For example, if `b` is to be an integer data type, consider replacing `a` and `c` with integer data types.

Padding in structures prohibits vectorization. In Example 3-17, there is no benefit in aligning every `a` component, because the NEON unit can load unaligned structures without penalty.

Example 3-17 Non vectorizable code caused by alignment padding

```
struct aligned_data
{
    char a;
    char b;
    char c;
    char not_used;
} n[10];
```

The code in Example 3-17 could be rewritten for vectorization by removing the `not_used` padding.

Example of improving performance by tuning source code

The compiler can provide diagnostic information to indicate where vectorization optimizations are successfully applied and where it failed to apply vectorization. See *--diag_suppress=optimizations* on page 2-47 and *--diag_warning=optimizations* on page 2-49 in the *Compiler Reference Guide*.

Example 3-18 shows two functions that implement a simple sum operation on an array. This code does not vectorize.

Example 3-18 Non vectorizable code

```
int addition(int a, int b)
{
    return a + b;
}
void add_int(int *pa, int *pb, unsigned int n, int x)
{
    unsigned int i;
    for(i = 0; i < n; i++) *(pa + i) = addition(*(pb + i),x);
}
```

Using the `--diag_warning=optimizations` option produces an optimization warning message for the `addition()` function.

Adding the __inline qualifier to the definition of `addition()` enables this code to vectorize but it is still not optimal. Using the `--diag_warning=optimizations` option again, produces optimization warning messages to indicate that the loop vectorizes but there might be a potential pointer aliasing problem.

The compiler must generate a runtime test for aliasing and output both vectorized and scalar copies of the code. Example 3-19 shows how this can be improved using the restrict keyword if you know that the pointers are not aliased.

Example 3-19 Using restrict to improve vectorization performance

```
__inline int addition(int a, int b)
{
    return a + b;
}
void add_int(int * __restrict pa, int * __restrict pb, unsigned int n, int x)
{
    unsigned int i;
    for(i = 0; i < n; i++) *(pa + i) = addition(*(pb + i),x);
}
```

The final improvement that can be made is to the number of loop iterations. In Example 3-19, the number of iterations is not fixed and might not be a multiple that can fit exactly into a NEON register. This means that the compiler must test for remaining iterations to execute using non vectored code. If you know that your iteration count is

one of those supported by NEON, you can indicate this to the compiler. Example 3-20 shows the final improvement that can be made to obtain the best performance from vectorization.

Example 3-20 Code tuned for best vectorization performance

```
__inline int addition(int a, int b)
{
    return a + b;
}
void add_int(int * __restrict pa, int * __restrict pb, unsigned int n, int x)
{
    unsigned int i;
    for(i = 0; i < (n & ~3); i++) *(pa + i) = addition(*(pb + i),x);
    /* n is a multiple of 4 */
}
```

Using __promise to improve vectorization

The `__promise(expr)` intrinsic is a promise to the compiler that a given expression is nonzero. This enables the compiler to improve vectorization by optimizing away code that, based on the promise you have made, is redundant.

The disassembled output of Example 3-21 shows the difference that `__promise` makes, reducing the disassembly to a simple vectorized loop by the removal of a scalar fix-up loop.

Example 3-21 Using `__promise(expr)` to improve vectorization code

```
void f(int *x, int n)
{
    int i;
    __promise((n > 0) && ((n&7)==0));
    for (i=0; i<n;i++) x[i]++;
}
```

Example 3-20 is a similar example that can benefit from the use of `__promise()`.

3.4 Examples

The following are examples of vectorizable code. See Example 3-22 and Example 3-23 on page 3-20.

The compiler options required to build these examples are `-O3 -Otime --vectorize -DNDEBUG --cpu=Cortex-A8` (or another processor that has NEON technology, such as Cortex-A9).

The use of `__promise` enables the compiler to generate smaller and faster code. See *Using __promise to improve vectorization* on page 3-17. The code works and vectorizes without `__promise`, but is then larger and slower.

Example 3-22 Vectorization code

```
/*
 * Vectorizable example code.
 * Copyright 2006 ARM. All rights reserved.
 *
 * Includes embedded assembly to initialize cpu; link using '--entry=init_cpu'.
 *
 * Build using:
 *   armcc --vectorize -c vector_example.c --cpu Cortex-A8 -Otime -O3 -DNDEBUG
 *   armlink -o vector_example.axf vector_example.o --entry=init_cpu
 */
#include <stdio.h>
#include <assert.h> /* for __promise() */
void fir(short *__restrict y, const short *x, const short *h, int n_out, int n_coefs)
{
    int n;
    /* I promise 'n_out is always a positive multiple of 8' */
    __promise(0 < n_out && (n_out % 8) == 0);
    for (n = 0; n < n_out; n++)
    {
        int k, sum = 0;
        /* I promise 'n_coefs is always a positive multiple of 4' */
        __promise(0 < n_coefs && (n_coefs % 4) == 0);
        for (k = 0; k < n_coefs; k++)
        {
            sum += h[k] * x[n - n_coefs + 1 + k];
        }
        y[n] = ((sum>>15) + 1) >> 1;
    }
}
int main()
{
    static const short x[128] =
```

```
{
    0x0000, 0x0647, 0x0c8b, 0x12c8, 0x18f8, 0x1f19, 0x2528, 0x2b1f,
    0x30fb, 0x36ba, 0x3c56, 0x41ce, 0x471c, 0x4c3f, 0x5133, 0x55f5,
    0x5a82, 0x5ed7, 0x62f2, 0x66cf, 0x6a6d, 0x6dca, 0x70e2, 0x73b5,
    0x7641, 0x7884, 0x7a7d, 0x7c29, 0x7d8a, 0x7e9d, 0x7f62, 0x7fd8,
    0x8000, 0x7fd8, 0x7f62, 0x7e9d, 0x7d8a, 0x7c29, 0x7a7d, 0x7884,
    0x7641, 0x73b5, 0x70e2, 0x6dca, 0x6a6d, 0x66cf, 0x62f2, 0x5ed7,
    0x5a82, 0x55f5, 0x5133, 0x4c3f, 0x471c, 0x41ce, 0x3c56, 0x36ba,
    0x30fb, 0x2b1f, 0x2528, 0x1f19, 0x18f8, 0x12c8, 0x0c8b, 0x0647,
    0x0000, 0xf9b9, 0xf375, 0xed38, 0xe708, 0xe0e7, 0xdad8, 0xd4e1,
    0xcf05, 0xc946, 0xc3aa, 0xbe32, 0xb8e4, 0xb3c1, 0xaecd, 0xaa0b,
    0xa57e, 0xa129, 0x9d0e, 0x9931, 0x9593, 0x9236, 0x8f1e, 0x8c4b,
    0x89bf, 0x877c, 0x8583, 0x83d7, 0x8276, 0x8163, 0x809e, 0x8028,
    0x8000, 0x8028, 0x809e, 0x8163, 0x8276, 0x83d7, 0x8583, 0x877c,
    0x89bf, 0x8c4b, 0x8f1e, 0x9236, 0x9593, 0x9931, 0x9d0e, 0xa129,
    0xa57e, 0xaa0b, 0xaecd, 0xb3c1, 0xb8e4, 0xbe32, 0xc3aa, 0xc946,
    0xcf05, 0xd4e1, 0xdad8, 0xe0e7, 0xe708, 0xed38, 0xf375, 0xf9b9,
};
static const short coeffs[8] =
{
    0x0800, 0x1000, 0x2000, 0x4000,
    0x4000, 0x2000, 0x1000, 0x0800
};
int i, ok = 1;
short y[128];
static const short expected[128] =
{
    0x1474, 0x1a37, 0x1fe9, 0x2588, 0x2b10, 0x307d, 0x35cc, 0x3afa,
    0x4003, 0x44e5, 0x499d, 0x4e27, 0x5281, 0x56a9, 0x5a9a, 0x5e54,
    0x61d4, 0x6517, 0x681c, 0x6ae1, 0x6d63, 0x6fa3, 0x719d, 0x7352,
    0x74bf, 0x6de5, 0x66c1, 0x5755, 0x379e, 0x379e, 0x5755, 0x66c1,
    0x6de5, 0x74bf, 0x7352, 0x719d, 0x6fa3, 0x6d63, 0x6ae1, 0x681c,
    0x6517, 0x61d4, 0x5e54, 0x5a9a, 0x56a9, 0x5281, 0x4e27, 0x499d,
    0x44e5, 0x4003, 0x3afa, 0x35cc, 0x307d, 0x2b10, 0x2588, 0x1fe9,
    0x1a37, 0x1474, 0x0ea5, 0x08cd, 0x02f0, 0xfd10, 0xf733, 0xf15b,
    0xeb8c, 0xe5c9, 0xe017, 0xda78, 0xd4f0, 0xcf83, 0xca34, 0xc506,
    0xbffd, 0xbb1b, 0xb663, 0xb1d9, 0xad7f, 0xa957, 0xa566, 0xa1ac,
    0x9e2c, 0x9ae9, 0x97e4, 0x951f, 0x929d, 0x905d, 0x8e63, 0x8cae,
    0x8b41, 0x8a1b, 0x893f, 0x88ab, 0x8862, 0x8862, 0x88ab, 0x893f,
    0x8a1b, 0x8b41, 0x8cae, 0x8e63, 0x905d, 0x929d, 0x951f, 0x97e4,
    0x9ae9, 0x9e2c, 0xa1ac, 0xa566, 0xa957, 0xad7f, 0xb1d9, 0xb663,
    0xbb1b, 0xbffd, 0xc506, 0xca34, 0xcf83, 0xd4f0, 0xda78, 0xe017,
    0xe5c9, 0xebcc, 0xf229, 0xf96a, 0x02e9, 0x0dd8, 0x1937, 0x24ce,
};
fir(y, x + 7, coeffs, 128, 8);
for (i = 0; i < sizeof(y)/sizeof(*y); ++i)
{
    if (y[i] != expected[i])
    {
        printf("mismatch: y[%d] = 0x%04x; expected[%d] = 0x%04x\n", i, y[i], i, expected[i]);
```

```
            ok = 0;
            break;
        }
    }
    if (ok) printf("** TEST PASSED OK **\n");
    return ok ? 0 : 1;
}
#ifdef __TARGET_ARCH_7_A
__asm void init_cpu() {
    // Set up CPU state
    MRC p15,0,r4,c1,c0,0
    ORR r4,r4,#0x00400000   // enable unaligned mode (U=1)
    BIC r4,r4,#0x00000002   // disable alignment faults (A=0)
    // MMU not enabled: no page tables
    MCR p15,0,r4,c1,c0,0
#ifdef __BIG_ENDIAN
    SETEND BE
#endif
    MRC p15,0,r4,c1,c0,2    // Enable VFP access in the CAR -
    ORR r4,r4,#0x00f00000   // must be done before any VFP instructions
    MCR p15,0,r4,c1,c0,2
    MOV r4,#0x40000000      // Set EN bit in FPEXC
    MSR FPEXC,r4
    IMPORT __main
    B __main
}
#endif
```

Example 3-23 DSP vectorization code

```
/*
* DSP Vectorizable example code.
* Copyright 2006 ARM. All rights reserved.
*
* Includes embedded assembly to initialize cpu; link using '--entry=init_cpu'.
*
* Build using:
*   armcc -c dsp_vector_example.c --cpu Cortex-A8 -Otime -O3 --vectorize -DNDEBUG
*   armlink -o dsp_vector_example.axf dsp_vector_example.o --entry=init_cpu
*/
#include <stdio.h>
#include <dspfns.h>
#include <assert.h> /* for __promise() */
void fn(short *__restrict r, int n, const short *__restrict a, const short *__restrict b)
{
    int i;
    /* I promise 'n is always a positive multiple of 8' */
```

```
    __promise(0 < n && (n % 8) == 0);
    for (i = 0; i < n; ++i)
    {
        r[i] = add(a[i], b[i]);
    }
}
int main()
{
    static const short x[128] =
    {
        0x0000, 0x0647, 0x0c8b, 0x12c8, 0x18f8, 0x1f19, 0x2528, 0x2b1f,
        0x30fb, 0x36ba, 0x3c56, 0x41ce, 0x471c, 0x4c3f, 0x5133, 0x55f5,
        0x5a82, 0x5ed7, 0x62f2, 0x66cf, 0x6a6d, 0x6dca, 0x70e2, 0x73b5,
        0x7641, 0x7884, 0x7a7d, 0x7c29, 0x7d8a, 0x7e9d, 0x7f62, 0x7fd8,
        0x8000, 0x7fd8, 0x7f62, 0x7e9d, 0x7d8a, 0x7c29, 0x7a7d, 0x7884,
        0x7641, 0x73b5, 0x70e2, 0x6dca, 0x6a6d, 0x66cf, 0x62f2, 0x5ed7,
        0x5a82, 0x55f5, 0x5133, 0x4c3f, 0x471c, 0x41ce, 0x3c56, 0x36ba,
        0x30fb, 0x2b1f, 0x2528, 0x1f19, 0x18f8, 0x12c8, 0x0c8b, 0x0647,
        0x0000, 0xf9b9, 0xf375, 0xed38, 0xe708, 0xe0e7, 0xdad8, 0xd4e1,
        0xcf05, 0xc946, 0xc3aa, 0xbe32, 0xb8e4, 0xb3c1, 0xaecd, 0xaa0b,
        0xa57e, 0xa129, 0x9d0e, 0x9931, 0x9593, 0x9236, 0x8f1e, 0x8c4b,
        0x89bf, 0x877c, 0x8583, 0x83d7, 0x8276, 0x8163, 0x809e, 0x8028,
        0x8000, 0x8028, 0x809e, 0x8163, 0x8276, 0x83d7, 0x8583, 0x877c,
        0x89bf, 0x8c4b, 0x8f1e, 0x9236, 0x9593, 0x9931, 0x9d0e, 0xa129,
        0xa57e, 0xaa0b, 0xaecd, 0xb3c1, 0xb8e4, 0xbe32, 0xc3aa, 0xc946,
        0xcf05, 0xd4e1, 0xdad8, 0xe0e7, 0xe708, 0xed38, 0xf375, 0xf9b9,
    };
    static const short y[128] =
    {
        0x8000, 0x7fd8, 0x7f62, 0x7e9d, 0x7d8a, 0x7c29, 0x7a7d, 0x7884,
        0x7641, 0x73b5, 0x70e2, 0x6dca, 0x6a6d, 0x66cf, 0x62f2, 0x5ed7,
        0x5a82, 0x55f5, 0x5133, 0x4c3f, 0x471c, 0x41ce, 0x3c56, 0x36ba,
        0x30fb, 0x2b1f, 0x2528, 0x1f19, 0x18f8, 0x12c8, 0x0c8b, 0x0647,
        0x0000, 0xf9b9, 0xf375, 0xed38, 0xe708, 0xe0e7, 0xdad8, 0xd4e1,
        0xcf05, 0xc946, 0xc3aa, 0xbe32, 0xb8e4, 0xb3c1, 0xaecd, 0xaa0b,
        0xa57e, 0xa129, 0x9d0e, 0x9931, 0x9593, 0x9236, 0x8f1e, 0x8c4b,
        0x89bf, 0x877c, 0x8583, 0x83d7, 0x8276, 0x8163, 0x809e, 0x8028,
        0x8000, 0x8028, 0x809e, 0x8163, 0x8276, 0x83d7, 0x8583, 0x877c,
        0x89bf, 0x8c4b, 0x8f1e, 0x9236, 0x9593, 0x9931, 0x9d0e, 0xa129,
        0xa57e, 0xaa0b, 0xaecd, 0xb3c1, 0xb8e4, 0xbe32, 0xc3aa, 0xc946,
        0xcf05, 0xd4e1, 0xdad8, 0xe0e7, 0xe708, 0xed38, 0xf375, 0xf9b9,
        0x0000, 0x0647, 0x0c8b, 0x12c8, 0x18f8, 0x1f19, 0x2528, 0x2b1f,
        0x30fb, 0x36ba, 0x3c56, 0x41ce, 0x471c, 0x4c3f, 0x5133, 0x55f5,
        0x5a82, 0x5ed7, 0x62f2, 0x66cf, 0x6a6d, 0x6dca, 0x70e2, 0x73b5,
        0x7641, 0x7884, 0x7a7d, 0x7c29, 0x7d8a, 0x7e9d, 0x7f62, 0x7fd8,
    };
    short r[128];
    static const short expected[128] =
    {
        0x8000, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff,
```

```
        0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff,
        0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff,
        0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff,
        0x8000, 0x7991, 0x72d7, 0x6bd5, 0x6492, 0x5d10, 0x5555, 0x4d65,
        0x4546, 0x3cfb, 0x348c, 0x2bfc, 0x2351, 0x1a90, 0x11bf, 0x08e2,
        0x0000, 0xf71e, 0xee41, 0xe570, 0xdcaf, 0xd404, 0xcb74, 0xc305,
        0xbaba, 0xb29b, 0xaaab, 0xa2f0, 0x9b6e, 0x942b, 0x8d29, 0x866f,
        0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000,
        0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000,
        0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000,
        0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000,
        0x8000, 0x866f, 0x8d29, 0x942b, 0x9b6e, 0xa2f0, 0xaaab, 0xb29b,
        0xbaba, 0xc305, 0xcb74, 0xd404, 0xdcaf, 0xe570, 0xee41, 0xf71e,
        0x0000, 0x08e2, 0x11bf, 0x1a90, 0x2351, 0x2bfc, 0x348c, 0x3cfb,
        0x4546, 0x4d65, 0x5555, 0x5d10, 0x6492, 0x6bd5, 0x72d7, 0x7991,
    };
    int i, ok = 1;
    fn(r, sizeof(r)/sizeof(*r), x, y);

    for (i = 0; i < sizeof(r)/sizeof(*r); ++i)
    {
        if (r[i] != expected[i])
        {
            printf("mismatch: r[%d] = 0x%04x; expected[%d] = 0x%04x\n", i, r[i], i, expected[i]);
            ok = 0;
            break;
        }
    }
    if (ok) printf("** TEST PASSED OK **\n");
        return ok ? 0 : 1;
}
#ifdef __TARGET_ARCH_7_A
__asm void init_cpu()
{
    // Set up CPU state
    MRC p15,0,r4,c1,c0,0
    ORR r4,r4,#0x00400000   // enable unaligned mode (U=1)
    BIC r4,r4,#0x00000002   // disable alignment faults (A=0)
    // MMU not enabled: no page tables
    MCR p15,0,r4,c1,c0,0
#ifdef __BIG_ENDIAN
    SETEND BE
#endif
    MRC p15,0,r4,c1,c0,2    // Enable VFP access in the CAR -
    ORR r4,r4,#0x00f00000   // must be done before any VFP instructions
    MCR p15,0,r4,c1,c0,2
    MOV r4,#0x40000000      // Set EN bit in FPEXC
    MSR FPEXC,r4
    IMPORT __main
```

```
    B __main
}
#endif
```

Chapter 4
Compiler Features

This chapter gives an overview of ARM-specific features of the compiler. It includes the following sections:

4.1 Intrinsics

The compiler supports several families of intrinsics, including:

- Instruction intrinsics for realizing ARM, Thumb, and NEON instructions from your C and C++ code
- Intrinsics realizing the ETSI basic operations
- Intrinsics emulating intrinsics found on the TI C55x compiler
- NEON™ intrinsics for use with the NEON vectorizing compiler.

This section describes these families of intrinsics.

4.1.1 About intrinsics

C and C++ are suited to a wide variety of tasks but do not provide inbuilt support for specific areas of application, for example, *Digital Signal Processing* (DSP).

Within a given application domain, there is usually a range of domain-specific operations that have to be performed frequently. Often, however, these operations cannot be efficiently implemented in C or C++. A typical example is the saturated add of two 32-bit signed two's complement integers, commonly used in DSP programming. Example 4-1 shows its implementation in C.

Example 4-1 C implementation of saturated add operation

```
#include <limits.h>
int L_add(const int a, const int b)
{
    int c;
    c = a + b;
    if (((a ^ b) & INT_MIN) == 0)
    {
        if ((c ^ a) & INT_MIN)
        {
            c = (a < 0) ? INT_MIN : INT_MAX;
        }
    }
    return c;
}
```

Intrinsic functions provide a way of easily incorporating domain-specific operations in C and C++ source code without resorting to complex implementations, for example, in embedded assembler or inline assembler. An intrinsic function has the appearance of a

function call in C or C++, but is replaced during compilation by a specific sequence of low-level instructions. When implemented using an intrinsic, for example, the saturated add function of Example 4-1 on page 4-2 has the form:

```
#include <dspfns.h>    /* Include ETSI intrinsics */
...
int a, b, result;
...
result = L_add(a, b);  /* Saturated add of a and b */
```

The use of intrinsics offers the following performance benefits:

- The low-level instructions substituted for an intrinsic might be more efficient than corresponding implementations in C or C++, resulting in both reduced instruction and cycle counts. To implement the intrinsic, the compiler automatically generates the best sequence of instructions for the specified target architecture. For example, the `L_add` intrinsic maps directly to the ARM v5TE assembly language instruction `qadd`:

  ```
  QADD r0, r0, r1    /* Assuming r0 = a, r1 = b on entry */
  ```

- More information is given to the compiler than the underlying C and C++ language is able to convey. This enables the compiler to perform optimizations and to generate instructions sequences that it could not otherwise have performed.

These performance benefits can be significant for real-time processing applications. However, care is required because the use of intrinsics can decrease code portability.

4.1.2 Instruction intrinsics

The ARM compiler provides a range of instruction intrinsics for realizing ARM assembly language instructions from within your C or C++ code. Collectively, these intrinsics enable you to emulate inline assembly code using a combination of C code and instruction intrinsics.

Generic intrinsics

The *Compiler Reference Guide* describes the following generic intrinsics that are ARM language extensions to the ISO C and C++ standards:

- *__breakpoint* on page 4-75
- *__current_pc* on page 4-78
- *__current_sp* on page 4-78
- *__nop* on page 4-89
- *__return_address* on page 4-95
- *__semihost* on page 4-97.

See also *GNU builtin functions* on page 4-195 in the *Compiler Reference Guide*.

Implementations of these intrinsics are available across all architectures.

Intrinsics for controlling IRQ and FIQ interrupts

The *Compiler Reference Guide* describes the following intrinsics that enable you to control IRQ and FIQ interrupts:

- *__disable_irq* on page 4-80
- *__enable_irq* on page 4-82
- *__disable_fiq* on page 4-79
- *__enable_fiq* on page 4-82.

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

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

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

  ```
  CPSID  i
  ```

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

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

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

  ```
  BL    __ARM_disable_irq
  ```

For more information on these instructions, see the *Assembler Guide*.

Intrinsics for inserting optimization barriers

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

The following optimization barrier intrinsics do not generate code, but they can result in slightly increased code size and additional memory accesses. In the *Compiler Reference Guide*, see:

- *__schedule_barrier* on page 4-96
- *__force_stores* on page 4-84
- *__memory_changed* on page 4-88.

Note

On some systems the memory barrier intrinsics might not be sufficient to ensure memory consistency. For example, the `__memory_changed()` intrinsic forces values held in registers to be written out to memory. However, if the destination for the data is held in a region that can be buffered it might wait in a write buffer. In this case you might also have to write to CP15 or use a memory barrier instruction to drain the write buffer. Refer to the Technical Reference Manual for your ARM processor for more information.

Intrinsics for inserting native instructions

The following intrinsics enable you to insert ARM processor instructions into the instruction stream generated by the compiler. In the *Compiler Reference Guide*, see:

- *__cdp* on page 4-76
- *__clrex* on page 4-77
- *__ldrex* on page 4-84
- *__ldrt* on page 4-87
- *__pld* on page 4-89
- *__pli* on page 4-91
- *__rbit* on page 4-94
- *__rev* on page 4-94
- *__ror* on page 4-96
- *__sev* on page 4-98
- *__strex* on page 4-101
- *__strt* on page 4-103
- *__swp* on page 4-105
- *__wfe* on page 4-107
- *__wfi* on page 4-107
- *__yield* on page 4-108.

Intrinsics for Digital Signal Processing

The following intrinsics described in the *Compiler Reference Guide* assist in the implementation of DSP algorithms:

- *__clz* on page 4-77
- *__fabs* on page 4-83
- *__fabsf* on page 4-84
- *__qadd* on page 4-92
- *__qdbl* on page 4-93
- *__qsub* on page 4-93
- *__sqrt* on page 4-99
- *__sqrtf* on page 4-100
- *__ssat* on page 4-100
- *__usat* on page 4-106.

See also *ARMv6 SIMD intrinsics* on page 4-109 in the *Compiler Reference Guide*.

These intrinsics introduce the appropriate target instructions for:

- ARM architectures from ARM v5TE onwards
- Thumb-2 architectures except 'M' variants.

Not every instruction has its own intrinsic. The compiler can combine several instrinsics, or combinations of intrinsics and C operators to generate more powerful instructions. For example, the ARM5TE `QDADD` instruction is realized by a combination of `__qadd` and `__qdbl`.

4.1.3 ETSI basic operations

The *European Telecommunications Standard Institute* (ETSI) has produced several recommendations for the coding of speech, for example, the G.723.1 and G.729 recommendations. These recommendations include source code and test sequences for reference implementations of the codecs.

Model implementations of speech codecs supplied by the ETSI are based on a collection of C functions known as the *ETSI basic operations*. The ETSI basic operations include 16-bit, 32-bit and 40-bit operations for saturated arithmetic, 16-bit and 32-bit logical operations, and 16-bit and 32-bit operations for data type conversion.

Note

Version 2.0 of the ETSI collection of basic operations, as described in the *ITU-T Software Tool Library 2005 User's manual*, introduces new 16-bit, 32-bit and 40 bit-operations. These operations are not supported in RVCT.

The ETSI basic operations serve as a set of primitives for developers publishing codec algorithms, rather than as a library for use by developers implementing codecs in C or C++. RVCT provides support for the ETSI basic operations through the header file `dspfns.h`.

ETSI operations in RVCT

The `dspfns.h` header file contains definitions of the ETSI basic operations as a combination of C code and intrinsics. RVCT supports the original ETSI family of basic operations described in the ETSI G.729 recommendation *Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)*, including:

- 16-bit and 32-bit saturated arithmetic operations, such as `add` and `sub`. For example, `add(v1, v2)` adds two 16-bit numbers `v1` and `v2` together, with overflow control and saturation, returning a 16-bit result.
- 16-bit and 32-bit multiplication operations, such as `mult` and `L_mult`. For example, `mult(v1, v2)` multiplies two 16-bit numbers `v1` and `v2` together, returning a scaled 16-bit result.
- 16-bit arithmetic shift operations, such as `shl` and `shr`. For example, the saturating left shift operation `shl(v1, v2)` arithmetically shifts the 16-bit input `v1` left `v2` positions. A negative shift count shifts `v1` right `v2` positions.
- 16-bit data conversion operations, such as `extract_l`, `extract_h`, and `round`. For example, `round(L_v1)` rounds the lower 16 bits of the 32-bit input `L_v1` into the most significant 16 bits with saturation.

  Note

  Beware that both the `dspfns.h` header file and the ISO C99 header file `math.h` both define (different versions of) the function `round()`. Take care to avoid this potential conflict.

See the header file `dspfns.h` for a complete list of the ETSI basic operations supported in RVCT.

In addition, see:

- ETSI Recommendation G.191: *Software tools for speech and audio coding standardization*
- *ITU-T Software Tool Library 2005 User's manual*, included as part of ETSI Recommendation G.191

- ETSI Recommendation G723.1: *Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s*
- ETSI Recommendation G.729: *Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP).*

These documents are available from ITU-T, the telecommunications bureau of the ITU, at `http://www.itu.int`.

Overflow and carry

The implementation of the ETSI basic operations in `dspfns.h` exposes the status flags `Overflow` and `Carry`. These flags are available as global variables for use in your own C or C++ programs. For example:

```
#include <dspfns.h>                 /* include ETSI intrinsics */
#include <stdio.h>
...
const int BUFLEN=255;
int a[BUFLEN], b[BUFLEN], c[BUFLEN];
...
Overflow = 0;                       /* clear overflow flag */
for (i = 0; i < BUFLEN; ++i) {
    c[i] = L_add(a[i], b[i]);       /* saturated add of a[i] and b[i] */
}
if (Overflow)
{
    fprintf(stderr, "Overflow on saturated addition\n");
}
```

Generally, saturating functions have a sticky effect on overflow. That is, the overflow flag remains set until it is explicitly cleared. For more information, see the header file `dspfns.h`.

4.1.4 TI C55x intrinsics

The *Texas Instruments* (TI) C55x compiler recognizes a number of intrinsics for the optimization of C code. RVCT supports the emulation of selected TI C55x intrinsics through the header file, `c55x.h`. TI C55x intrinsics that are emulated in `c55x.h` include:

- Intrinsics for addition, subtraction, negation and absolute value, such as `_sadd` and `_ssub`. For example, `_sadd(v1, v2)` returns the 16-bit saturated sum of `v1` and `v2`.
- Intrinsics for multiplication and shifting, such as `_smpy` and `_sshl`. For example, `_smpy(v1, v2)` returns the saturated fractional-mode product of `v1` and `v2`.

- Intrinsics for rounding, saturation, bitcount and extremum, such as `_round` and `_count`. For example, `_round(v1)` returns the value `v1` rounded by adding 215 using unsaturated arithmetic, clearing the lower 16 bits.

The following TI C55x intrinsics are not supported in `c55x.h`:

- Associative variants of intrinsics for addition and multiply-and-accumulate. This includes all TI C55x intrinsics prefixed with `_a_`, for example, `_a_sadd` and `_a_smac`.
- Rounding variants of intrinsics for multiplication and shifting, for example, `_smacr` and `_smasr`.
- All **`long long`** variants of intrinsics. This includes all TI C55x intrinsics prefixed with `_ll`, for example, `_llsadd` and `_llshl`. **`long long`** variants of intrinsics are not supported in RVCT because they operate on 40-bit data.
- All arithmetic intrinsics with side effects. For example, the TI C55x intrinsics `_firs` and `_lms` are not defined in `c55x.h`.
- Intrinsics for ETSI support functions, such as `L_add_c` and `L_sub_c`.

  Note

  An exception is the ETSI support function for saturating division, `divs`. This intrinsic is supported in `c55x.h`.

See the header file `c55x.h` for a complete list of the TI C55x intrinsics emulated in RVCT.

For more information on TI compiler intrinsics see `http://www.ti.com`.

4.1.5 NEON Intrinsics

The ARM compiler provides NEON intrinsics to provide an intermediate step for SIMD code generation between a vectorizing compiler and writing assembler code. This feature makes it easier to write code that takes advantage of the NEON architecture when compared to writing assembler directly.

The NEON intrinsics are defined in the header file `arm_neon.h`. The header file defines both the intrinsics and a set of vector types. See Appendix E *Using NEON Support* in the *Compiler Reference Guide* for more information about NEON intrinsics.

Example 4-2 on page 4-10 shows a short example using NEON intrinsics. To build the example:

1. Compile the C file `neon_example.c` with the following options:

```
armcc -c --debug --cpu=Cortex-A8 neon_example.c
```

2. Link the image using the command:

```
armlink neon_example.o -o neon_example.axf
```

3. Use a compatible debugger, for example RealView Debugger, to load and run the image.

Example 4-2 NEON intrinsics

```
/* neon_example.c - Neon intrinsics example program */
#include <stdint.h>
#include <stdio.h>
#include <assert.h>
#include <arm_neon.h>
/* fill array with increasing integers beginning with 0 */
void fill_array(int16_t *array, int size)
{    int i;
    for (i = 0; i < size; i++)
    {
         array[i] = i;
    }
}
/* return the sum of all elements in an array. This works by calculating 4
totals (one for each lane) and adding those at the end to get the final total */
int sum_array(int16_t *array, int size)
{
     /* initialize the accumulator vector to zero */
     int16x4_t acc = vdup_n_s16(0);
     int32x2_t acc1;
     int64x1_t acc2;
     /* this implementation assumes the size of the array is a multiple of 4 */
     assert((size % 4) == 0);
     /* counting backwards gives better code */
     for (; size != 0; size -= 4)
     {
          int16x4_t vec;
          /* load 4 values in parallel from the array */
          vec = vld1_s16(array);
          /* increment the array pointer to the next element */
          array += 4;
          /* add the vector to the accumulator vector */
          acc = vadd_s16(acc, vec);
      }
      /* calculate the total */
      acc1 = vpaddl_s16(acc);
      acc2 = vpaddl_s32(acc1);
      /* return the total as an integer */
```

```
      return (int)vget_lane_s64(acc2, 0);
}
/* main function */
int main()
{
      int16_t my_array[100];
      fill_array(my_array, 100);
      printf("Sum was %d\n", sum_array(my_array, 100));
      return 0;
}
```

For more information about NEON see:

- *NEON technology* on page 2-23
- The *Assembler Guide*.

4.2 Named register variables

The compiler enables you to access registers of an ARM architecture-based processor using named register variables.

Named register variables are declared by combining the **`register`** keyword with the `__asm` keyword. The `__asm` keyword takes one parameter, a character string, that names the register. For example, the declaration:

```
register int R0 __asm("r0");
```

declares `R0` as a named register variable for the register `r0`. See *Named register variables* on page 4-192 in the *Compiler Reference Guide* for more information on the registers of ARM architecture-based processors that can be accessed using named register variables.

You can declare named register variables as global variables. You can declare some, but not all, named register variables as local variables. In general, do not declare *Vector Floating-Point* (VFP) registers and core registers as local variables. Do not declare caller-save registers, such as R0, as local variables. (Caller-save registers are registers that the caller must save the values of, if it wants the values after the subroutine completes.) Your program might still compile if you declare these locally, but you risk unexpected runtime behavior if you do this.

A typical use of named register variables is to access bits in the *Application Program Status Register* (APSR) (see *The Application Program Status Register (APSR)* on page 2-7 in the *Assembler Guide*). Example 3-3 shows the use of named register variables to set the saturation flag `Q` in the APSR.

Example 4-3 Setting bits in the APSR using a named register variable

```
#ifndef __BIG_ENDIAN // bitfield layout of APSR is sensitive to endianness
typedef union
{
    struct
    {
        int mode:5;
        int T:1;
        int F:1;
        int I:1;
        int _dnm:19;
        int Q:1;
        int V:1;
        int C:1;
        int Z:1;
        int N:1;
```

```
    } b;
    unsigned int word;
} PSR;
#else /* __BIG_ENDIAN */
typedef union
{
    struct
    {
        int N:1;
        int Z:1;
        int C:1;
        int V:1;
        int Q:1;
        int _dnm:19;
        int I:1;
        int F:1;
        int T:1;
        int mode:5;
    } b;
    unsigned int word;
} PSR;
#endif /* __BIG_ENDIAN */
register PSR apsr __asm("apsr");
void set_Q(void)
{
    apsr.b.Q = 1;
}
```

4.3 Pragmas

The ARM compiler recognizes the following forms of pragma:

```
#pragma no_feature-name

#pragma feature_name
```

Note

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

For more information see the relevant section in the *Compiler Reference Guide*:

Pragmas for saving and restoring the pragma state

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

- *#pragma pop* on page 4-70
- *#pragma push* on page 4-70.

Pragmas controlling optimization goals

These pragmas enable you to assign optimization goals to individual functions. The pragmas must be placed outside of a function. The following pragmas control these optimizations:

- *#pragma Onum* on page 4-67
- *#pragma Ospace* on page 4-68
- *#pragma Otime* on page 4-68.

Pragmas controlling code generation

The following pragmas control how code is generated:

- *#pragma arm* on page 4-59
- *#pragma thumb* on page 4-73
- *#pragma exceptions_unwind, #pragma no_exceptions_unwind* on page 4-64.

Pragmas controlling loop unrolling:

The following pragmas control how loops are unrolled:

- *#pragma unroll [(n)]* on page 4-71
- *#pragma unroll_completely* on page 4-72.

Pragmas controlling *PreCompiled Header* (PCH) processing

The following pragmas control PCH processing:

- *#pragma hdrstop* on page 4-64

- *#pragma no_pch* on page 4-67.

Pragmas controlling anonymous structures and unions

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

- *#pragma anon_unions, #pragma no_anon_unions* on page 4-58.

Pragmas controlling diagnostic messages

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

- *#pragma diag_default tag[,tag,...]* on page 4-61
- *#pragma diag_error tag[,tag,...]* on page 4-62
- *#pragma diag_remark tag[,tag,...]* on page 4-62
- *#pragma diag_suppress tag[,tag,...]* on page 4-63
- *#pragma diag_warning tag[, tag, ...]* on page 4-64.

Miscellaneous pragmas

- *#pragma arm section [section_sort_list]* on page 4-59
- *#pragma import(__use_full_stdio)* on page 4-65
- *#pragma inline, #pragma no_inline* on page 4-66
- *#pragma once* on page 4-67
- *#pragma pack(n)* on page 4-68
- *#pragma softfp_linkage, #pragma no_softfp_linkage* on page 4-70
- *#pragma import symbol_name* on page 4-65.

4.4 Bit-banding

This section describes how the bit-banding feature is supported by the compiler.

Note

Bit-banding is a feature of the Cortex-M3 processor and some derivatives. This functionality is not available on other ARM processors.

For more information on architectural support for bit-banding, see the *Technical Reference Manual* for your processor.

Bit-banding is supported in the following ways:

- `__attribute__((bitband))` language extension
- `--bitband` command-line option.

4.4.1 Using `__attribute__((bitband))`

`__attribute__((bitband))` is a type attribute that is used to bit-band type definitions of structures. See Example 4-4 and Example 4-5 on page 4-17.

Example 4-4 Unplaced object

```
/* foo.c */

typedef struct {
  int i : 1;
  int j : 2;
  int k : 3;
} BB __attribute__((bitband));

BB value; // Unplaced object

void update_value(void)
{
    value.i = 1;
    value.j = 0;
}

/* end of foo.c */
```

In Example 4-4 on page 4-16 the unplaced bit-banded objects must be relocated into the bit-band region. You can do this by either using an appropriate scatter-loading description file or by using the `--rw_base` linker command-line option. See the *Linker Reference Guide* for more information.

Alternatively, you can use `__attribute__((at()))` to place bit-banded objects at a particular address in the bit-band region. See Example 4-5.

Example 4-5 Placed object

```
/* foo.c */

typedef struct {
  int i : 1;
  int j : 2;
  int k : 3;
} BB __attribute__((bitband));

BB value __attribute__((at(0x20000040))); // Placed object

void update_value(void)
{
    value.i = 1;
    value.j = 0;
}

/* end of foo.c */
```

See *__attribute__((bitband))* on page 4-43 and *__attribute__((at(address)))* on page 4-48 in the *Compiler Reference Guide* for more information.

4.4.2 Using `--bitband` on the command-line

The `--bitband` command-line option bit-bands all non **`const`** global structure objects.

When `--bitband` is applied to `foo.c` in Example 4-6 on page 4-18, the write to `value.i` is bit-banded, that is, value `0x00000001` is written to the bit-band alias word that `value.i` maps to in the bit-band region.

Accesses to `value.j` and `value.k` are not bit-banded.

Example 4-6 Using `--bitband` command-line option

```
/* foo.c */

typedef struct {
  int i : 1;
  int j : 2;
  int k : 3;
} BB;

BB value __attribute__((at(0x20000040))); // Placed object

void update_value(void)
{
    value.i = 1;
    value.j = 0;
}

/* end of foo.c */
```

`armcc` supports the bit-banding of objects accessed through absolute addresses. When `--bitband` is applied to `foo.c` in Example 4-7, the access to `rts` is bit-banded.

Example 4-7 Bit-banding of objects accessed through absolute addresses

```
/* foo.c */

typedef struct {
  int rts : 1;
  int cts : 1;
  unsigned int data;
} uart;

#define com2 (*((volatile uart *)0x20002000))
void put_com2(int n)
{
  com2.rts = 1;
  com2.data = n;
}

/* end of foo.c */
```

See *--bitband* on page 2-18 in the *Compiler Reference Guide* for more information.

4.4.3 Restrictions

The following restrictions apply:

- Bit-banding can only be used with **`struct`** types. Any union type or other aggregate type with a union as a member cannot be bit-banded.
- Members of structs cannot be bit-banded individually.
- Bit-banded accesses are generated only for single-bit bitfields.
- Bit-banded accesses are not generated for **`const`** objects, pointers, and local objects.

4.5 Thread-local storage

Thread-Local Storage (TLS) is a class of static storage that, like the stack, is private to each thread of execution. Each thread in a process is given a location where it can store thread-specific data. Variables are allocated so that there is one instance of the variable for each existing thread.

Before each thread terminates, it releases its dynamic memory and any pointers to thread-local variables in that thread become invalid.

See *__declspec(thread)* on page 4-30 in the *Compiler Reference Guide*.

4.6 Eight-byte alignment features

The ARM compiler has the following eight-byte alignment features:

- The *Procedure Call Standard for the ARM Architecture* (AAPCS) requires that the stack is eight-byte aligned at all external interfaces. The ARM compiler and C libraries preserve the eight-byte alignment of the stack. In addition, the default C library memory model maintains eight-byte alignment of the heap.
- Code is compiled in a way that requires and preserves the eight byte alignment constraints at external interfaces.
- If you have assembly files, or legacy objects, or libraries in your project, it is your responsibility to check that they preserve eight-byte stack alignment, and correct them if required. See the *Assembler Guide* and the *Linker User Guide*.
- In RVCT v2.0 and later, **`double`** and **`long long`** data types are eight-byte aligned. This enables efficient use of the `LDRD` and `STRD` instructions in ARMv5TE and later.
- The default implementations of `malloc()`, `realloc()`, and `calloc()` maintain an eight-byte aligned heap.
- The default implementation of `alloca()` returns an eight-byte aligned block of memory. See *alloca()* on page 2-68 in the *Libraries Guide* for more information on this C library extension.

Chapter 5
Coding Practices

The ARM compiler `armcc` is a mature, industrial-strength ISO C and C++ compiler capable of producing highly optimized, high quality machine code. By using programming practices and techniques that work well on RISC processors such as ARM cores, however, you can increase the portability, efficiency and robustness of your C and C++ source code. This chapter describes some of these programming practices, together with some programming techniques that are specific to ARM processors.

This chapter includes the following sections:

5.1 Optimizing code

The ARM compiler is highly optimizing, for small code size and high performance. The compiler performs optimizations common to other optimizing compilers, for example, data-flow optimizations such as common sub-expression elimination and loop optimizations such as loop combining and distribution. In addition, the compiler performs a range of optimizations specific to ARM architecture-based processors.

Even though the compiler is highly optimizing, you can often significantly improve the performance of your C or C++ code by selecting correct optimization criteria, target processor and architecture, and inlining options.

5.1.1 Optimizing for size versus speed

The compiler provides two options for optimizing for code size and performance:

`-Ospace` This option causes the compiler to optimize mainly for code size. This is the default option.

`-Otime` This option causes the compiler to optimize mainly for speed.

For best results, you must build your application using the most appropriate command-line option.

Note

These command-line options instruct the compiler to use optimizations that deliver the effect wanted in the vast majority of cases. However, it is not guaranteed that `-Otime` always generates faster code, or that `-Ospace` always generates smaller code.

See:

- *-Ospace* on page 2-99 in the *Compiler Reference Guide*
- *-Otime* on page 2-99 in the *Compiler Reference Guide*.

5.1.2 Optimization levels and the debug view

The precise optimizations performed by the compiler depend both on the level of optimization chosen, and whether you are optimizing for performance or code size.

The compiler supports the following optimization levels:

`-O0` Minimum optimization. The compiler performs simple optimizations that do not impair the debug view.

When debugging is enabled, this option gives the best possible debug view.

`-O1` Restricted optimization.

When debugging is enabled, this option gives a generally satisfactory debug view with good code density.

`-O2` High optimization. This is the default optimization level.

When debugging is enabled, this option might give a less satisfactory debug view.

`-O3` Maximum optimization. This is the most aggressive form of optimization available. Specifying this option enables multifile compilation by default where multiple files are specified on the command line.

When debugging is enabled, this option typically gives a poor debug view.

Because optimization affects the mapping of object code to source code, the choice of optimization level and `-Ospace`/`-Otime` generally impacts the debug view. When debugging is enabled using `--debug`, explicitly specify the most appropriate optimization level using the `-Onum` command-line option.

The option `-O0` is the best option to use if a simple debug view is needed. Selecting `-O0` typically increases the size of the ELF image by 7-15%. To reduce the size of your debug tables, use the `--no_debug_macros` option.

See:

- *--debug, --no_debug* on page 2-37 in the *Compiler Reference Guide*
- *--debug_macros, --no_debug_macros* on page 2-37 in the *Compiler Reference Guide*
- *--dwarf2* on page 2-51 in the *Compiler Reference Guide*
- *--dwarf3* on page 2-51 in the *Compiler Reference Guide*
- *-Onum* on page 2-96 in the *Compiler Reference Guide*.

5.1.3 Selecting the target CPU

Each new version of the ARM architecture typically supports extra instructions, extra modes of operation, pipeline differences, and register renaming.

- Where a compiled program is to run on a specific ARM architecture-based processor, it is best to select the target processor using the `--cpu` command-line option. This enables the compiler to make full use of instructions that are supported by the processor, and also to perform processor-specific optimizations such as instruction scheduling.

- Where a compiled program is to run on different ARM processors, you must choose the lowest common denominator architecture appropriate for your application using the `--cpu` command-line option. For example, to compile code for processors supporting the ARM v6 architecture, use the command-line option `--cpu 6`.

Note

You can list all the processors and architectures supported by the compiler using the command-line option `--cpu list`.

See:

- *Specifying the target processor or architecture* on page 2-23
- *--cpu=list* on page 2-30 in the *Compiler Reference Guide*
- *--cpu=name* on page 2-30 in the *Compiler Reference Guide*.

5.1.4 Optimizing loops

Loops are a common construct in most programs. Because a significant amount of execution time is often spent in loops, it is worthwhile paying attention to time-critical loops.

Loop termination

The loop termination condition can cause significant overhead if written without caution. Where possible:

- always write count-down-to-zero loops and use simple termination conditions
- always use a counter of type **`unsigned int`**, and test for not equal to zero.

Table 5-1 shows two sample implementations of a routine to calculate `n!` that together illustrate loop termination overhead. The first implementation calculates `n!` using an incrementing loop, while the second routine calculates `n!` using a decrementing loop.

Table 5-1 C code for incrementing and decrementing loops

Incrementing loop	Decrementing loop

`int fact1(int n)`
`{`
`  int i, fact = 1;`
`  for (i = 1; i <= n; i++)`
`  fact *= i;`
`  return (fact);`
`}`
	`int fact2(int n)`
	`{`
	`  unsigned int i, fact = 1;`
	`  for (i = n; i != 0; i--)`
	`  fact *= i;`
	`  return (fact);`
	`}`

Table 5-2 shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations of Table 5-1 on page 5-4, where the C code for both implementations has been compiled using the options `-O2 -Otime`.

Table 5-2 C Disassembly for incrementing and decrementing loops

Incrementing loop	Decrementing loop

`fact1 PROC`
`  MOV  r2, r0`
`  MOV  r0, #1`
`  CMP  r2, #1`
`  MOV  r1, r0`
`  BXLT  lr`
`|L1.20|`
`  MUL  r0, r1, r0`
`  ADD  r1, r1, #1`
`  CMP  r1, r2`
`  BLE  |L1.20|`
`  BX  lr`
`  ENDP`
	`fact2 PROC`
	`  MOVS  r1, r0`
	`  MOV  r0, #1`
	`  BXEQ  lr`
	`|L1.12|`
	`  MUL  r0, r1, r0`
	`  SUBS  r1, r1, #1`
	`  BNE  |L1.12|`
	`  BX  lr`
	`ENDP`

Comparing the disassemblies of Table 5-2 shows that the `ADD/CMP` instruction pair in the incrementing loop disassembly has been replaced with a single `SUBS` instruction in the decrementing loop disassembly. This is because a compare with zero can be optimized away.

In addition to saving an instruction in the loop, the variable `n` does not have to be saved across the loop, so the use of a register is also saved in the decrementing loop disassembly. This eases register allocation.

The technique of initializing the loop counter to the number of iterations required, and then decrementing down to zero, also applies to **`while`** and **`do`** statements.

Loop unrolling

Small loops can be unrolled for higher performance, with the disadvantage of increased code size. When a loop is unrolled, a loop counter needs to be updated less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled, so that the loop overhead completely disappears. The ARM compiler unrolls loops automatically at `-O3 -Otime`. Otherwise, any unrolling must be done in source code.

Note

Manual unrolling of loops might hinder the automatic re-rolling of loops and other loop optimizations by the compiler.

The advantages and disadvantages of loop unrolling can be illustrated using the two sample routines shown in Table 5-3. Both routines efficiently test a single bit by extracting the lowest bit and counting it, after which the bit is shifted out.

The first implementation uses a loop to count bits. The second routine is the first unrolled four times, with an optimization applied by combining the four shifts of `n` into one. Unrolling frequently provides new opportunities for optimization.

Table 5-3 C code for rolled and unrolled bit-counting loops

Bit-counting loop

```
int countbit1(unsigned int n)
{
    int bits = 0;
    while (n != 0)
    {
        if (n & 1) bits++;
        n >>= 1;
    }
    return bits;
}
```

Unrolled bit-counting loop

```
int countbit2(unsigned int n)
{
    int bits = 0;
    while (n != 0)
    {
        if (n & 1) bits++;
        if (n & 2) bits++;
        if (n & 4) bits++;
        if (n & 8) bits++;
        n >>= 4;
    }
    return bits;
}
```

Table 5-4 shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations of Table 5-3 on page 5-6, where the C code for each implementation has been compiled using the option -O2.

Table 5-4 Disassembly for rolled and unrolled bit-counting loops

Bit-counting loop	Unrolled bit-counting loop

countbit1 PROC
  MOV r1, #0
  B |L1.20|
|L1.8|
  TST r0, #1
  ADDNE r1, r1, #1
  LSR r0, r0, #1
|L1.20|
  CMP r0, #0
  BNE |L1.8|
  MOV r0, r1
  BX lr
ENDP
	countbit2 PROC
	  MOV r1, r0
	  MOV r0, #0
	  B |L1.48|
	|L1.12|
	  TST r1, #1
	  ADDNE r0, r0, #1
	  TST r1, #2
	  ADDNE r0, r0, #1
	  TST r1, #4
	  ADDNE r0, r0, #1
	  TST r1, #8
	  ADDNE r0, r0, #1
	  LSR r1, r1, #4
	|L1.48|
	  CMP r1, #0
	  BNE |L1.12|
	  BX lr
	ENDP

On the ARM7, checking a single bit takes six cycles in the disassembly of the bit-counting loop shown in the leftmost column. The code size is only nine instructions. The unrolled version of the bit-counting loop checks four bits at a time, taking on average only three cycles per bit. However, the cost is the larger code size of fifteen instructions.

5.1.5 Using volatile

Occasionally, you might encounter problems when compiling code at the higher optimization levels -O2 and -O3. For example, you might get stuck in a loop when polling hardware or multi-threaded code might exhibit strange behavior. In such cases it is likely that you have to declare some of your variables as **volatile**.

Declaring a variable as **volatile** tells the compiler that the variable can be modified at any time externally to the implementation, for example, by the operating system or by hardware. Because the value of a **volatile**-qualified variable can change at any time, the physical address of the variable in memory must always be accessed whenever the variable is referenced in code. This means the compiler cannot perform optimizations on the variable, for example, caching it in a local register to avoid memory accesses.

In contrast, when a variable is not declared as **volatile**, the compiler can assume its value cannot be modified outside the implementation. Therefore, the compiler can perform optimizations on the variable.

The use of the **volatile** keyword is illustrated in the two sample routines of Table 5-5, both of which loop reading a buffer until a status flag `buffer_full` is set to true. Both routines assume that the state of `buffer_full` can change asynchronously with program flow.

The first routine shows a naive implementation of the loop. Notice that the variable `buffer_full` is not qualified as **volatile** in this implementation. In contrast, the second routine shows the same loop where `buffer_full` is correctly qualified as **volatile** in the implementation.

Table 5-5 C code for nonvolatile and volatile buffer loops

Nonvolatile version of buffer loop	Volatile version of buffer loop

`int buffer_full;`
`int read_stream(void)`
`{`
`  int count = 0;`
`  while (!buffer_full)`
`  {`
`  count++;`
`  }`
`  return count;`
`}`
	`volatile int buffer_full;`
	`int read_stream(void)`
	`{`
	`  int count = 0;`
	`  while (!buffer_full)`
	`  {`
	`  count++;`
	`  }`
	`  return count;`
	`}`

Table 5-6 shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations of Table 5-1 on page 5-4, where the C code for each implementation has been compiled using the option -O2.

Table 5-6 Disassembly for nonvolatile and volatile buffer loop

Nonvolatile version of buffer loop	Volatile version of buffer loop

`read_stream PROC`
`    LDR      r1, |L1.28|`
`    MOV      r0, #0`
`    LDR      r1, [r1, #0]`
`|L1.12|`
`    CMP      r1, #0`
`    ADDEQ    r0, r0, #1`
`    BEQ      |L1.12|      ; infinite loop`
`    BX       lr`
`    ENDP`
`|L1.28|`
`    DCD      ||.data||`
`    AREA ||.data||, DATA, ALIGN=2`
`buffer_full`
`    DCD      0x00000000`
	`read_stream PROC`
	`    LDR      r1, |L1.28|`
	`    MOV      r0, #0`
	`|L1.8|`
	`    LDR      r2, [r1, #0];  ; buffer_full`
	`    CMP      r2, #0`
	`    ADDEQ    r0, r0, #1`
	`    BEQ      |L1.8|`
	`    BX       lr`
	`    ENDP`
	`|L1.28|`
	`    DCD      ||.data||`
	`    AREA ||.data||, DATA, ALIGN=2`
	`    buffer_full`
	`        DCD      0x00000000`

In the disassembly of the nonvolatile version of the buffer loop in Table 5-6, the statement `LDR r0, [r0, #0]` loads the value of `buffer_full` into register `r0` outside the loop labeled `|L1.8|`. Because `buffer_full` is not declared as **`volatile`**, the compiler assumes that its value cannot be modified outside the program. Having already read the value of `buffer_full` into `r0`, the compiler omits reloading the variable when optimizations are enabled, because its value cannot change. The result is the infinite loop labeled `|L1.8|`.

In contrast, in the disassembly of the volatile version of the buffer loop, the compiler assumes the value of `buffer_full` can change outside the program and performs no optimizations. Consequently, the value of `buffer_full` is loaded into register `r0` inside the loop labeled `|L1.4|`. As a result, the loop `|L1.4|` is implemented correctly in assembly code.

To avoid optimization problems caused by changes to program state external to the implementation, you must declare variables as **`volatile`** whenever their values can change unexpectedly in ways unknown to the implementation. In practice, you must declare a variable as **`volatile`** whenever you are:

- accessing memory mapped peripherals
- sharing global variables between multiple threads
- accessing global variables in an interrupt routine.

5.2 Code metrics

Code metrics provide a means of objectively evaluating code quality. The ARM compiler, linker, and profiler provide several facilities for generating simple code metrics and improving code quality. In particular, you can:

- measure code and data sizes
- generate static callgraphs
- measure stack use
- reduce debug information in objects and libraries.

See the *ARM Profiler User Guide* for information about the ARM Profiler.

5.2.1 Measuring code and data sizes

You can measure the code and data sizes of your application using a range of options. See:

- *--info=totals* on page 2-74 in the *Compiler Reference Guide*
- *--info=topic[,topic,...]* on page 2-36 in the *Utilities Guide*
- *--callgraph, --no_callgraph* on page 2-18 in the *Linker Reference Guide*
- *--map, --no_map* on page 2-59 in the *Linker Reference Guide*
- *--symbols, --no_symbols* on page 2-89 in the *Linker Reference Guide*
- *--xref, --no_xref* on page 2-101 in the *Linker Reference Guide*.

5.2.2 Measuring stack use

C and C++ both use the stack intensively. For example, the stack is used to hold:

- the return address of functions
- registers that must be preserved, as determined by the AAPCS
- local variables, including local arrays, structures, and, in C++, classes.

In general, there is no way to automatically measure stack use. However, it is possible to manually estimate the extent of stack utilization. This can be done in several ways:

- Link with `--callgraph` to produce a static callgraph. This shows information on all functions, including stack use.
- Link with `--info=stack` or `--info=summarystack` to list the stack usage of all global symbols.
- Use your debugger to set a watchpoint on the last available location in the stack and see if the watchpoint is ever hit.

- Use your debugger to:
  1. Allocate space for the stack that is much larger than you expect to require.
  2. Fill the stack with a known value, for example, zero or 0xDEADDEAD.
  3. Run your application, or a fixed portion of it. Aim to use as much of the stack as possible in the test run. For example, be sure to execute as many branches of your code as possible, and to generate interrupts where appropriate, so that they are included in the stack trace.
  4. Examine, after your application has finished executing, the stack area of memory to see how many of the known values (zeros or 0xDEADDEAD) have been overwritten. The stack shows garbage in the part of the stack that has been used and zeros or 0xDEADDEAD values in the remainder.
  5. Count the number of known entries and multiply by eight. This shows how far the stack has grown in memory in bytes.
- For RVISS, use a map file to define a region of memory where access is not permitted. Place this region directly below your stack in memory. If the stack overflows into the forbidden region, a data abort occurs that can be trapped by your debugger.

See:

- *Measuring code and data sizes* on page 5-10
- *--callgraph, --no_callgraph* on page 2-18 of the *Linker Reference Guide*.

5.2.3 Reducing debug information in objects and libraries

It is often useful to reduce the amount of debug information in objects and libraries. Reducing the level of debug information:

- Reduces the size of objects and libraries, thereby reducing the amount of disk space needed to store them.
- Speeds up link time. In the compilation cycle, most of the link time is consumed by reading in all the debug sections and eliminating the duplicates.
- Minimizes the size of the final image. This facilitates the fast loading and processing of debug symbols by a debugger.

There are several ways to reduce the amount of debug information being generated per source file. For example, you can:

- Avoid conditional use of `#define` in header files. This might make it more difficult for the linker to eliminate duplicate information.

- Modify your C or C++ source files so that header files are `#included` in the same order.
- Partition header information into smaller blocks. That is, use a larger number of smaller header files rather than a smaller number of larger header files. This helps the linker to eliminate more of the common blocks.
- Only include a header file in a C or C++ source file if it is really needed.
- Guard against the multiple inclusion of header files. For example, if you have a header file `foo.h`, then add:

  ```
  #ifndef foo_h
  #define foo_h
  ...
  // rest of header file as before
  ...
  #endif /* foo_h */
  ```

  You can use the compiler option `--remarks` to warn about unguarded header files.
- Compile your code with the `--no_debug_macros` command-line option to discard preprocessor macro definitions from debug tables.

See:

- *--debug_macros, --no_debug_macros* on page 2-37 in the *Compiler Reference Guide*
- *--remarks* on page 2-111 in the *Compiler Reference Guide*.

5.3 Functions

To enable the compiler to perform optimizations more efficiently, it is a good idea in general to keep functions small and simple. There are several ways of achieving this goal. For example, you can:

- minimize the number of parameters passed to and from functions
- return multiple values from a function through the registers using `__value_in_regs`
- where possible, qualify functions as `__pure`.

5.3.1 Minimizing parameter passing overhead

There are several ways in which you can minimize the overhead of passing parameters to functions. For example:

- Ensure that functions take four or fewer arguments if each argument is a word or less in size. In C++, ensure that nonstatic member functions take three or fewer arguments, because of the implicit `this` pointer argument that is usually passed in `R0`.
- Ensure that a function does a significant amount of work if it requires more than four arguments, so that the cost of passing the stacked arguments is outweighed.
- Put related arguments in a structure, and pass a pointer to the structure in any function call. This reduces the number of parameters and increases readability.
- Minimize the number of **`long long`** parameters, because these take two argument words that have to be aligned on an even register index.
- Minimize the number of **`double`** parameters if software floating-point is enabled.
- Avoid functions with a variable number of parameters. Functions taking a variable number of arguments effectively pass all their arguments on the stack.

5.3.2 __value_in_regs

In C and C++, one way of returning multiple values from a function is to use a structure. Normally, structures are returned on the stack, with all the associated expense this entails.

To reduce memory traffic and reduce code size, the compiler enables you to return multiple values from a function through the registers. Up to four words can be returned from a function in a **`struct`** by qualifying the function with `__value_in_regs`. For example:

```
typedef struct s_coord { int x; int y; } coord;
coord reflect(int x1, int y1) __value_in_regs;
```

You can use `__value_in_regs` anywhere where you have to return multiple values from a function. Examples include:

- returning multiple values from C and C++ functions
- returning multiple values from embedded assembly language functions
- making supervisor calls
- re-implementing `__user_initial_stackheap`.

See *__value_in_regs* on page 4-20 in the *Compiler Reference Guide* for more information about `__value_in_regs`.

5.3.3 __pure

A pure function is a function that always returns the same result if it is called with the same arguments.

By definition, it is sufficient to evaluate any particular call to a pure function only once. Because the result of a call to the function is guaranteed to be the same for any identical call, each subsequent call to the function in code can be replaced with the result of the original call.

To instruct the compiler that a function is pure, declare the function as `__pure`.

The use of the `__pure` keyword is illustrated in the two sample routines of Table 5-7 on page 5-15. Both routines call a function `fact` to calculate the sum of `n!` and `n!`. The `fact` function depends only on its input argument n to compute `n!`. Therefore `fact` is a pure function.

The first routine shows a naive implementation of the function fact, where fact is not declared __pure. In the second implementation, the function fact is qualified as __pure to indicate to the compiler that it is a pure function.

Table 5-7 C code for pure and impure functions

A pure function not declared __pure	A pure function declared __pure

`int fact(int n)`
`{`
`  int f = 1;`
`  while (n > 0)`
`  f *= n--;`
`  return f;`
`}`
`int foo(int n)`
`{`
`  return fact(n)+fact(n);`
`}`
	`int fact(int n) __pure`
	`{`
	`  int f = 1;`
	`  while (n > 0)`
	`  f *= n--;`
	`  return f;`
	`}`
	`int foo(int n)`
	`{`
	`  return fact(n)+fact(n);`
	`}`

Table 5-8 shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations of Table 5-7, where the C code for each implementation has been compiled using the option -O2.

Table 5-8 Disassembly for pure and impure functions

A pure function not declared __pure	A pure function declared __pure

`fact PROC`
`  ...`
`foo PROC`
`  MOV  r3, r0`
`  PUSH  {lr}`
`  BL  fact`
`  MOV  r2, r0`
`  MOV  r0, r3`
`  BL  fact`
`  ADD  r0, r0, r2`
`  POP  {pc}`
`  ENDP`
	`fact PROC`
	`  ...`
	`foo PROC`
	`  PUSH  {lr}`
	`  BL  fact`
	`  LSL  r0,r0,#1`
	`  POP  {pc}`
	`  ENDP`

In the disassembly of the function foo in Table 4-8 where fact is not qualified as __pure, the function fact is called twice, because the compiler does not know the function is a candidate for CSE. In contrast, in the disassembly of foo in Table 4-8 where fact is qualified as __pure, fact is called only once, instead of twice, because the compiler has been able to perform CSE when adding fact(n) + fact(n).

By definition, pure functions cannot have side effects. For example, a pure function cannot read or write global state by using global variables or indirecting through pointers, because accessing global state can violate the rule that the function must return the same value each time when called twice with the same parameters. Therefore, you must use `__pure` carefully in your programs. Where functions can be declared `__pure`, however, the compiler can often perform powerful optimizations, such as CSEs.

See *__pure* on page 4-13 in the *Compiler Reference Guide* for more information about pure functions.

5.3.4 Placing ARM function qualifiers

Many ARM keyword extensions modify the behavior or calling sequence of a function. For example, `__pure`, `__irq`, `__swi`, `__swi_indirect`, `__softfp`, and `__value_in_regs` all behave in this way.

These function modifiers all have a common syntax. A function modifier such as `__pure` can qualify a function declaration either:

- Before the function declaration. For example:

  ```
  __pure int foo(int);
  ```

- After the closing parenthesis on the parameter list. For example:

  ```
  int foo(int) __pure;
  ```

For simple function declarations, each syntax is unambiguous. However, for a function whose return type or arguments are function pointers, the prefix syntax is imprecise. For example, the following function returns a function pointer, but it is not clear whether `__pure` modifies the function itself or its returned pointer type:

```
__pure int (*foo(int)) (int); /* declares 'foo' as a (pure?) function that
                                 returns a pointer to a (pure?) function.
                                 It is ambiguous which of the two function
                                 types is pure. */
```

In fact, the single `__pure` keyword at the front of the declaration of `foo` modifies both `foo` itself *and* the function pointer type returned by `foo`.

In contrast, the postfix syntax because it enables a clear distinction between whether `__pure` applies to the argument, the return type, or the base function, when declaring a function whose argument and return types are function pointers. For example:

```
int (*foo1(int) __pure) (int);       /* foo1 is a pure function returning
                                        a pointer to a normal function */
int (*foo2(int)) (int) __pure;       /* foo2 is a function returning
```

```
                                          a pointer to a pure function */
int (*foo3(int) __pure) (int) __pure; /* foo3 is a pure function returning
                                          a pointer to a pure function */
```

In this example:

- `foo1` and `foo3` are modified themselves
- `foo2` and `foo3` return a pointer to a modified function
- the functions `foo3` and `foo` are identical.

Because the postfix syntax is more precise than the prefix syntax, it is recommended that, where possible, you make use of the postfix syntax when qualifying functions with ARM function modifiers.

5.4 Function inlining

Function inlining offers a trade-off between code size and performance. By default, the compiler decides for itself whether to inline code or not. As a general rule, the compiler makes sensible decisions about inlining with a view to producing code of minimal size. This is because code size for embedded systems is of fundamental importance.

In most circumstances, the decision to inline a particular function is best left to the compiler. However, you can give the compiler a hint that a function is required to be inlined by using the appropriate inline keyword. The compiler also offers a range of other facilities for modifying its behavior with respect to inlining. There are several factors you must take into account when deciding whether to use these facilities, or more generally, whether to inline a function at all.

Functions that are qualified with `__inline`, `inline`, or `__forceinline` are called inline functions. In C++, member functions that are defined inside a class, struct, or union, are also inline functions.

Note

Be aware that profile guided optimizations can affect function inlining. See *--profile=filename* on page 2-107 in the *Compiler Reference Guide*.

5.4.1 How the compiler decides to inline

When inlining is enabled, the compiler uses a complex decision tree to decide when a function is inlined. The compiler uses the following simplified algorithm to determine if a function is to be inlined:

1. If the function is qualified with `__forceinline`, then the function is inlined if it is possible to do so.
2. If the function is qualified with `__inline` and the option `--forceinline` is selected, then the function is inlined if it is possible to do so.

   If the function is qualified with `__inline` and the option `--forceinline` is not selected, then the function is inlined if it is practical to do so.
3. If the optimization level is `-O2` or higher, or `--autoinline` is selected, then the function is inlined if it is practical to inline the function, and if it is possible to do so.

When deciding if it is practical to inline a function, the compiler takes into account several other criteria, including whether you select `-Ospace` or `-Otime`. Select `-Otime` to increase the likelihood that a function is inlined. See *When is it practical for the compiler to inline?* on page 5-19.

You cannot override any decision made by the compiler about when it is practical to inline a function. For example, you cannot force a function to be inlined if the compiler thinks it is not sensible.

5.4.2 When is it practical for the compiler to inline?

The compiler decides for itself when it is practical to inline a function or not, depending on a number of conditions, including:

- the size of the function, and how many times it is called
- the current optimization level
- whether it is optimizing for speed (`-Otime`) or size (`-Ospace`)
- whether the function has external or static linkage
- how many parameters the function has
- whether the return value of the function is used.

Ultimately, the compiler can decide not to inline a function, even if the function is qualified with `__forceinline`. As a general rule:

- smaller functions stand a better chance of being inlined
- compiling with `-Otime` increases the likelihood that a function is inlined
- large functions are not normally inlined because this can adversely affect code density and performance.

5.4.3 Managing inlining

You can force the compiler to attempt to inline a function using the `__forceinline` keyword. The compiler places the function inline, unless doing so causes problems. For example, a recursive function is inlined into itself only once. To force the compiler to attempt to inline all functions marked with `__inline`, compile your code with the `--forceinline` command-line option.

At the highest levels of optimization (`-O2` and `-O3`), the compiler is able to automatically inline functions if it is sensible to do so, even if you do not explicitly give a hint. See *Marking functions as static* on page 5-23.

You can control the automatic inlining of functions at the highest optimization levels using the `--no_autoinline` and `--autoinline` command-line options. In general, when automatic inlining is enabled, the compiler inlines anything that it is sensible to inline. When automatic inlining is disabled, only functions marked as `__inline` are candidates for inlining.

You can control whether inlining is performed at all using the `--no_inline` and `--inline` keywords. By default, inlining of functions is enabled. If you disable inlining of functions using the `--no_inline` command-line option, then the compiler attempts to inline only those functions that are explicitly qualified with `__forceinline`.

See:

- *--autoinline, --no_autoinline* on page 2-17 in the *Compiler Reference Guide*
- *--forceinline* on page 2-58 in the *Compiler Reference Guide*
- *--inline, --no_inline* on page 2-75 in the *Compiler Reference Guide*
- *__forceinline* on page 4-6 in the *Compiler Reference Guide*
- *__inline* on page 4-9 in the *Compiler Reference Guide*.

5.4.4 Automatic inlining

At `-O2` and `-O3` levels of optimization, the compiler considers inlining calls to functions that are defined, but are not inline functions. This works best for static functions, because if all uses of a static function can be inlined, no out-of-line copy is required. It is best to mark all non-inline functions as static if they are not used outside the translation unit where they are defined. A translation unit is the preprocessed output of a source file together with all of the headers and source files included as a result of the `#include` directive. Typically, you do not want to place definitions of non-inline functions in header files.

If you are compiling with the `--multifile` option, enabled by default at `-O3` level, or `--ltcg`, it is possible for the compiler to perform automatic inlining for calls to functions that are defined in other translation units.

For `--multifile`, both translation units must be compiled in the same invocation of the compiler. For `--ltcg`, they are required only to be linked together.

`--no_inline` disables automatic inlining.

5.4.5 Differences in behavior between C++, C90, C99, and GNU C90 compiler modes

The effect of inlining differs in some areas, depending on what language the compiler is compiling for.

`__forceinline` behaves like `__inline`, except that the compiler tries harder to do the inlining.

C++ and C90 mode

The **`inline`** keyword is not available in C90.

The effect of **__inline** in C90, and **__inline** and **inline** in C++, is identical.

When declaring an `extern` function to be inline, you must define it in every translation unit that it is used in. You must ensure that you use the same definition in each translation unit.

The requirement of defining the function in every translation unit applies even though it has external linkage.

If an inline function is used by more than one file, its definition is typically placed in a header file.

Placing definitions of non-inline functions in header files is not recommended, because this can result in the creation of a separate function in each translation unit. If the non-inline function is an **extern** function, this leads to duplicate symbols at link time. If the non-inline function is **static**, this can lead to unwanted code duplication.

Member functions defined within a C++ structure, class, or union declaration, are implicitly inline. They are treated as if they are declared with the **inline** or **__inline** keyword.

Inline functions have **extern** linkage unless they are explicitly declared **static**. If an inline function is declared to be static, any out-of-line copies of the function must be unique to their translation unit, so declaring an inline function to be static could lead to unwanted code duplication.

The compiler generates a regular call to an out-of-line copy of a function when it cannot inline the function, and when it decides not to inline it.

The requirement of defining a function in every translation unit it is used in means that the compiler is not required to emit out-of-line copies of all **extern** inline functions. When the compiler does emit out-of-line copies of an **extern** inline function, it uses Common Groups, so that the linker eliminates duplicates, keeping at most one copy in the same out-of-line function from different object files. (See *Common group or section elimination* on page 3-13 in the *Linker User Guide*).

C99 mode

The rules for C99 inline functions with external linkage differ to those of C++. C99 distinguishes between inline definitions and external definitions. Within a given translation unit where the inline function is defined, if the inline function is always declared with **inline** and never with **extern**, it is an inline definition. Otherwise, it is an external definition. These inline definitions are not used to generate out-of-line copies, even when `--no_inline` is used.

Each use of an inline function might be inlined using a definition from the same translation unit (that might be an inline definition or an external definition), or it might become a call to an external definition. If an inline function is used, it must have exactly one external definition in some translation unit. This is the same rule that applies to using any external function. In practise, if all uses of an inline function are inlined, no error occurs if the external definition is missing. If you use `--no_inline`, only external definitions are used.

Typically, you put inline functions with external linkage into header files as inline definitions, using **`inline`**, not **`extern`**. There is also an external definition in a source file. For example:

Example 5-1 Function inlining in C99

```
/* example_header.h */
inline int my_function (int i)
{
   return i + 42; // inline definition
}

/* file1.c */
#include “example_header.h”
   ... // uses of my_function()

/* file2.c */
#include “example_header.h”
   ... // uses of my_function()

/* myfile.c */
#include “example_header.h”
extern inline int my_function(int); // causes external definition.
```

This is the same strategy that is typically used for C++, but in C++, there is no special external definition, and no requirement for it.

The definitions of inline functions can be different in different translation units. However, in typical use, like *Function inlining in C99*, they are identical.

When compiling with `--multifile` or `--ltcg`, calls in one translation unit might be inlined using the external definition in another translation unit.

C99 places some restrictions on inline definitions. They cannot define modifiable local static objects. They cannot reference identifiers with static linkage.

In C99 mode, as with all other modes, the effects of **`__inline`** and **`inline`** are identical.

Inline functions with static linkage have the same behavior in C99 as in C++.

GNU C90 mode

The GNU C90 rules for inlining differ from the rules in other compiler modes. See the GNU documentation at `http://gcc.gnu.org`.

5.4.6 Linker inlining

The linker is able to replace calls to some very short functions with the body of the function. See *--inline, --no_inline* on page 2-47 in the *Linker Reference Guide*.

5.4.7 Debugging data and the --no_inline and --inline command-line options

The debug view generated for uses of inline functions is generally good. However, it is sometimes useful to avoid inlining functions because in some situations, debugging is clearer if they are not inlined. You can enable and disable the inlining of functions using the `--no_inline` and `--inline` command-line options. See *--inline, --no_inline* on page 2-75 in the *Compiler Reference Guide*.

5.4.8 Marking functions as static

At the optimization levels `-O2` and `-O3`, the compiler is able to automatically inline a function if it is practical to do so, even when the function is not declared as `__inline` or **`inline`**.

Note

To control the automatic inlining of functions at higher optimization levels, use the `--no_autoinline` and `--autoinline` command-line options.

Unless a function is explicitly declared as **`static`** (or `__inline`), the compiler has to retain the out-of-line version of it in the object file in case it is called from some other module. The linker is unable to remove unused out-of-line functions from an object unless you place them in their own sections using one of the following methods:

- *--split_sections* on page 2-118 in the *Compiler Reference Guide*
- *__attribute__((section("name")))* on page 4-52 in the *Compiler Reference Guide*
- *#pragma arm section [section_sort_list]* on page 4-59 in the *Compiler Reference Guide*
- linker feedback.

If you fail to declare functions that are never called from outside a module as **static**, your code can be adversely affected. In particular, you might have:

- A larger code size, because out-of-line versions of functions are retained in the image.

  When a function is automatically inlined, both the in-line version *and* an out-of-line version of the function might end up in the final image, unless the function is declared as **static**. This might possibly increase code size.

- An unnecessarily complicated debug view, because there are both inline versions and out-of-line versions of functions to display.

  Retaining both inline and out-of-line copies of a function in code can sometimes be confusing when setting breakpoints or single-stepping in a debug view. The debugger has to display both in-line and out-of-line versions in its interleaved source view, so that you can see what is happening when stepping through either the in-line or out-of-line version.

Because of these problems, declare non-inline functions as **static** when you are sure that they can never be called from another module.

5.4.9 Setting breakpoints on inline functions in ROM images

When you set a breakpoint on an inline function, the RealView Debugger attempts to set a breakpoint on each inlined instance of the function. If you are using RealView ICE to debug an image in ROM, and the number of inline instances is greater than the number of available hardware breakpoints, the debugger might not be able to set the additional breakpoints. In this case the debugger reports an error.

See:

- *--autoinline, --no_autoinline* on page 2-17 in the *Compiler Reference Guide*
- *--forceinline* on page 2-58 in the *Compiler Reference Guide*
- *--inline, --no_inline* on page 2-75 in the *Compiler Reference Guide*
- *__forceinline* on page 4-6 in the *Compiler Reference Guide*
- *__inline* on page 4-9 in the *Compiler Reference Guide*.

5.5 Aligning data

The various C data types are aligned on specific byte boundaries to maximize storage potential and to provide for fast, efficient memory access with the ARM instruction set. For example, the ARM architecture can access a four-byte variable using only one instruction when the object is stored at an address divisible by four, so four-byte objects are located on four-byte boundaries.

By default, the compiler stores data objects as shown in Table 5-9.

Table 5-9 Compiler storage of data objects by byte alignment

Type	Bytes	Alignment
`char`	1	Located at any byte address.
`short`	2	Located at any address that is evenly divisible by 2.
`float`, `int`, `long`, `pointer`	4	Located at an address that is evenly divisible by 4.
`long long` `double`	8	Located at an address that is evenly divisible by 4.

Data alignment becomes relevant when the compiler locates variables to physical memory addresses. For example, in the following structure, a three-byte gap is required between `bmem` and `cmem`.

```
struct example_st {
  int amem;
  char bmem;
  int cmem;
};
```

ARM and Thumb processors are designed to efficiently access naturally-aligned data, that is, doublewords that lie on addresses that are multiples of four, words that lie on addresses that are multiples of four, halfwords that lie on addresses that are multiples of two, and single bytes that lie at any byte address. Such data is located on its natural size boundary.

5.5.1 Types of data alignment

All accesses to data in memory can be classified into the following categories:

- Natural alignment, for example, on a word boundary at 0x1000. The ARM compiler normally aligns variables and pads structures so that these items are accessed efficiently using `LDR` and `STR` instructions.
- Known but non-natural alignment, for example, a word at address 0x1001. This type of alignment commonly occurs when structures are packed to remove unnecessary padding. In C and C++, the `__packed` qualifier or the `#pragma pack(n)` pragma is used to signify that a structure is packed.
- Unknown alignment, for example, a word at an arbitrary address. This type of alignment commonly occurs when defining a pointer that can point to a word at any address. In C and C++, the `__packed` qualifier or the `#pragma pack(n)` pragma is used to signify that a pointer can access a word on a non-natural alignment boundary.

See *The __packed qualifier and unaligned data access* on page 5-27 for more information about the `__packed` qualifier, packed structures, and unaligned pointers.

See *#pragma pack(n)* on page 4-68 in the *Compiler Reference Guide* for information about `#pragma pack(n)`.

5.5.2 Unaligned data access

It can be necessary to access unaligned data in memory, for example, when porting legacy code from a CISC architecture where instructions are available to directly access unaligned data in memory.

On ARMv4 and ARMv5 architectures, and on the ARMv6 architecture depending on how it is configured, care needs to be taken when accessing unaligned data in memory, lest unexpected results are returned. For example, when a conventional pointer is used to read a word in C or C++ source code, the ARM compiler generates assembly language code that reads the word using an `LDR` instruction. This works as expected when the address is a multiple of four, for example if it lies on a word boundary. However, if the address is not a multiple of four, the `LDR` returns a rotated result rather than performing a true unaligned word load. Generally, this rotation is not what the programmer expects.

On ARMv6 and later architectures, unaligned access is fully supported.

5.5.3 The __packed qualifier and unaligned data access

The `__packed` qualifier sets the alignment of any valid type to one. This enables objects of packed type to be read or written using unaligned accesses.

Examples of objects that can be packed include:

- structures
- unions
- pointers.

See *__packed* on page 4-11 in the *Compiler Reference Guide* for more information on the `__packed` qualifier.

Unaligned fields in structures

For efficiency, fields in a structure are located on their natural size boundary. This means that the compiler often inserts padding between fields to ensure they are aligned.

When space is at a premium, the `__packed` qualifier can be used to create structures without padding between fields. Structures can be packed in two ways:

- The entire **`struct`** can be declared as `__packed`. For example:

  ```
  __packed struct mystruct
  {
      char c;
      short s;
  } // not recommended
  ```

  Each field of the structure inherits the `__packed` qualifier.

  Declaring an entire **`struct`** as `__packed` typically incurs a penalty both in code size and performance. See *__packed structures versus individually __packed fields* on page 5-28.

- Individual non-aligned fields within the **`struct`** can be declared as `__packed`. For example:

  ```
  struct mystruct
  {
      char c;
      __packed short s; // recommended
  }
  ```

  This is the recommended approach to packing structures. See *__packed structures versus individually __packed fields* on page 5-28.

Note

The same principles apply to unions. You can declare either an entire union as `__packed`, or use the `__packed` attribute to identify components of the union that are unaligned in memory.

Reading from and writing to structures qualified with `__packed` requires unaligned accesses and can therefore incur a performance penalty. See *__packed structures versus individually __packed fields*.

Unaligned pointers

By default, the ARM compiler expects conventional C pointers to point to an aligned word in memory, because this enables the compiler to generate more efficient code.

If you want to define a pointer that can point to a word at any address, then you must specify this using the `__packed` qualifier when defining the pointer. For example:

```
__packed int *pi; // pointer to unaligned int
```

When a pointer is declared as `__packed`, the ARM compiler generates code that correctly accesses the dereferenced value of the pointer, regardless of its alignment. The generated code consists of a sequence of byte accesses, or variable alignment-dependent shifting and masking instructions, rather than a simple `LDR` instruction. Consequently, declaring a pointer as `__packed` incurs a performance and code size penalty.

Unaligned LDR instructions for accessing halfwords

In some circumstances the compiler might intentionally generate unaligned `LDR` instructions. In particular, the compiler can do this to load halfwords from memory, even where the architecture supports dedicated halfword load instructions.

For example, to access an unaligned **`short`** within a `__packed` structure, the compiler might load the required halfword into the top half of a register and then shift it down to the bottom half. This operation requires only one memory access, whereas performing the same operation using `LDRB` instructions requires two memory accesses, plus instructions to merge the two bytes.

5.5.4 __packed structures versus individually __packed fields

When optimizing a **`struct`** that is packed, the compiler tries to deduce the alignment of each field, to improve access. However, it is not always possible for the compiler to deduce the alignment of each field in a `__packed` **`struct`**. In contrast, when individual fields in a struct are declared as `__packed`, fast access is guaranteed to naturally aligned

members within the **struct**. Therefore, when the use of a packed structure is required, it is recommended that you always pack individual fields of the structure, rather than the entire structure itself.

Note

Declaring individual non-aligned fields of a **struct** as `__packed` also has the advantage of making it clearer to the programmer which fields of the **struct** are non-aligned.

The differences between not packing a **struct**, packing an entire **struct**, and packing individual fields of a **struct** are illustrated by the three implementations of a **struct** shown in Table 5-10.

In the first implementation, the **struct** is not packed. In the second implementation, the entire structure `mystruct` is qualified as `__packed`. In the third implementation, the `__packed` attribute is removed from the `mystruct` structure, and individual non-aligned fields are declared as `__packed`.

Table 5-10 C code for an unpacked struct, a packed struct, and a struct with individually packed fields

Unpacked struct	__packed struct	__packed fields

`struct foo`
`{`
`  char one;`
`  short two;`
`  char three;`
`  int four;`
`} c;`
	`__packed struct foo`
	`{`
	`  char one;`
	`  short two;`
	`  char three;`
	`  int four;`
	`} c;`
		`struct foo`
		`{`
		`  char one;`
		`  __packed short two;`
		`  char three;`
		`  int four;`
		`} c;`

Table 5-11 on page 5-30 shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations of Table 5-10, where the C code for each implementation has been compiled using the option `-O2`.

Note

The `-Ospace` and `-Otime` compiler options control whether accesses to unaligned elements are made inline or through a function call. Using `-Otime` results in inline unaligned accesses, while using `-Ospace` results in unaligned accesses made through function calls.

Table 5-11 Disassembly for an unpacked struct, a packed struct, and a struct with individually packed fields

Unpacked struct	__packed struct	__packed fields

`; r0 contains address of c`
`; char one`
`LDRB r1, [r0, #0]`
`; short two`
`LDRSH r2, [r0, #2]`
`; char three`
`LDRB r3, [r0, #4]`
`; int four`
`LDR r12, [r0, #8]`
	`; r0 contains address of c`
	`; char one`
	`LDRB r1, [r0, #0]`
	`; short two`
	`LDRB r2, [r0, #1]`
	`LDRSB r12, [r0, #2]`
	`ORR r2, r12, r2, LSL #8`
	`; char three`
	`LDRB r3, [r0, #3]`
	`; int four`
	`ADD r0, r0, #4`
	`BL __aeabi_uread4`
		`; r0 contains address of c`
		`; char one`
		`LDRB r1, [r0, #0]`
		`; short two`
		`LDRB r2, [r0, #1]`
		`LDRSB r12, [r0, #2]`
		`ORR r2, r12, r2, LSL #8`
		`; char three`
		`LDRB r3, [r0, #3]`
		`; int four`
		`LDR r12, [r0, #4]`

In the disassembly of the unpacked **struct** in Table 5-11, the compiler always accesses data on aligned word or halfword addresses. The compiler is able to do this because the **struct** is padded so that every member of the **struct** lies on its natural size boundary.

In the disassembly of the `__packed` **struct** in Table 5-11, the fields `one` and `three` are aligned on their natural size boundaries by default, and so the compiler makes aligned accesses. The compiler always carries out aligned word or halfword accesses for fields it can identify are aligned. For the unaligned field `two`, the compiler uses multiple aligned memory accesses (`LDR`/`STR`/`LDM`/`STM`), combined with fixed shifting and masking, to access the correct bytes in memory. The compiler calls the AEABI runtime routine `__aeabi_uread4` for reading an unsigned word at an unknown alignment to access the field `four`, because it is not able to determine that the field lies on its natural size boundary.

In the disassembly of the **struct** with individually packed fields in Table 5-11, the fields `one`, `two`, and `three` are accessed in the same way as in the case where the entire **struct** is qualified as `__packed`. In contrast to the situation where the entire **struct** is packed, however, the compiler makes a word-aligned access to the field `four`, because the presence of the `__packed short` within the structure helps the compiler to determine that the field `four` lies on its natural size boundary.

5.6 Using floating-point arithmetic

The ARM compiler provides many features for managing floating-point arithmetic both in software and in hardware. For example, you can specify software or hardware support for floating-point, particular hardware architectures, and the level of conformance to IEEE floating-point standards.

The selection of floating-point options determines various trade-offs between floating-point performance, system cost, and system flexibility. To obtain the best trade-off between performance, cost, and flexibility, you have to make sensible choices in your selection of floating-point options.

5.6.1 Support for floating-point operations

The ARM processor core does not contain floating-point hardware. Floating-point arithmetic must be supported separately, either:

- In software, through the floating-point library `fplib`. This library provides functions that can be called to implement floating-point operations using no additional hardware. See *The software floating-point library, fplib* on page 4-2 in the *Libraries Guide*.
- In hardware, using a hardware VFP coprocessor with the ARM processor core to provide the required floating-point operations. VFP is a coprocessor architecture that implements IEEE floating-point and supports single and double precision, but not extended precision.

  Note

  In practice, floating-point arithmetic in the VFP is implemented using a combination of hardware, that executes the common cases, and software, that deals with the uncommon cases, and cases causing exceptions. See *VFP support* on page 5-34.

The differences between software and hardware support for floating-point arithmetic are illustrated with Example 5-2, that shows a function implementing floating-point arithmetic operations in C.

Example 5-2 Floating-point operations

```
float foo(float num1, float num2)
{
    float temp, temp2;
    temp = num1 + num2;
```

```
    temp2 = num2 * num2;
    return temp2-temp;
}
```

When the C code of Example 5-2 on page 5-31 is compiled with the command-line option `--cpu 5TE --fpu softvfp`, the compiler produces machine code with the disassembly of Example 5-3. In this example, floating-point arithmetic is performed in software through calls to library routines such as `__aeabi_fmul`.

Example 5-3 Support for floating-point operations in software

```
||foo|| PROC
    PUSH     {r4-r6, lr}
    MOV      r4, r1
    BL       __aeabi_fadd
    MOV      r5, r0
    MOV      r1, r4
    MOV      r0, r4
    BL       __aeabi_fmul
    MOV      r1, r5
    POP      {r4-r6, lr}
    B        __aeabi_fsub
    ENDP
```

When the C code of Example 5-2 on page 5-31 is compiled with the command-line option `--fpu vfp`, the compiler produces machine code with the disassembly of Example 5-4. In this example, floating-point arithmetic is performed in hardware through floating-point arithmetic instructions such as `VMUL.F32`.

Example 5-4 Support for floating-point operations in hardware

```
||foo|| PROC
    VADD.F32 s2, s0, s1
    VMUL.F32 s0, s1, s1
    VSUB.F32 s0, s0, s2
    BX       lr
    ENDP
```

In practice, code that makes use of hardware support for floating-point arithmetic is more compact and offers better performance than code that performs floating-point arithmetic in software. However, hardware support for floating-point arithmetic requires a VFP coprocessor.

By default, if a VFP coprocessor is present, VFP instructions are generated. If there is no VFP coprocessor, the compiler generates code that makes calls to the software floating-point library `fplib` to carry out floating-point operations. `fplib` is available as part of the standard distribution of the RealView Development Suite of C libraries.

5.6.2 VFP architectures

VFP is a floating-point architecture that provides both single and double precision operations. Many operations can take place in either scalar form or in vector form. Several versions of the architecture are supported, including:

- VFPv2, implemented in:
  - the VFP10 revision 1, as provided by the ARM10200E
  - the VFP9-S, available as a separately licensable option for ARM926E/946E/966E
  - the VFP11, as provided in the ARM1136JF-S, ARM1176JZF-S, and ARM11 MPCore.
- VFPv3, implemented on ARM architecture v7 and later, for example, the Cortex-A8. VFPv3 is backwards compatible with VFPv2 except that it cannot trap floating point exceptions. It requires no software support code. VFPv3 has 32 double-precision registers.
- VFPv3 optionally extended with half-precision extensions. These extensions provide conversion functions between half-precision floating-point numbers and single-precision floating-point numbers, in both directions. They can be implemented with any Advanced SIMD and VFP implementation that supports single-precision floating-point numbers.
- VFPv3-D16 is an implementation of VFPv3 that provides 16 double-precision registers. It is implemented on ARM architecture v7 processors that support VFP without NEON.
- VFPv3U is an implementation of VFPv3 that can trap floating-point exceptions. It requires software support code.

Note

Particular implementations of the VFP architecture might provide additional implementation-specific functionality. For example, the VFP coprocessor hardware might include extra registers for describing exceptional conditions. This extra functionality is known as *sub-architecture* functionality. For more information about sub-architecture functionality, see *ARM Application Note 133 - Using VFP with RVDS*.

You can find this application note in the vfpsupport sub-directory of the Examples directory of your RealView Development Suite distribution at *install_directory*\RVDS\Examples\...\vfpsupport.

5.6.3 VFP support

ARM VFP coprocessors are optimized to process well-defined floating-point code in hardware. Arithmetic operations that occur too rarely, or that are too complex, are not handled in hardware. Instead, processing of these cases must be handled in software. This approach minimizes the amount of coprocessor hardware required and reduces costs.

Code provided to handle cases the VFP hardware is unable to process is known as VFP support code. When the VFP hardware is unable to deal with a situation directly, it bounces the case to VFP support code for more processing. For example, VFP support code might be called to process any of the following:

- floating-point operations involving NaNs
- floating-point operations involving denormals.
- floating-point overflow
- floating-point underflow
- inexact results
- division-by-zero errors
- invalid operations.

When support code is in place, the VFP supports a fully IEEE 754-compliant floating-point model.

Using VFP support

For convenience, an implementation of VFP support code that can be used in your system is provided with your installation of RVCT. The support code comprises:

- The libraries vfpsupport.l and vfpsupport.b for emulating VFP operations bounced by the hardware.

  These files are located in the \lib\armlib subdirectory of your RVCT installation.

- C source code and assembly language source code implementing top-level, second-level and user-level interrupt handlers.

  These files can be found in the vfpsupport subdirectory of the Examples directory of your RealView Development Suite distribution at *install_directory*\RVDS\Examples\...\vfpsupport.

These files might require modification to integrate VFP support with your operating system.

- C source code and assembly language source code for accessing subarchitecture functionality of VFP coprocessors.

  These files are located in the `vfpsupport` subdirectory of the `Examples` directory of your RealView Development Suite distribution at *`install_directory`*`\RVDS\Examples\...\vfpsupport`.

When the VFP coprocessor bounces an instruction, an Undefined Instruction exception is signaled to the processor and the VFP support code is entered through the Undefined Instruction vector. The top-level and second-level interrupt handlers perform some initial processing of the signal, for example, ensuring that the exception is not caused by an illegal instruction. The user-level interrupt handler then calls the appropriate library function in the library `vfpsupport.l` or `vfpsupport.b` to emulate the VFP operation in software.

Note

You do not have to use VFP support code:

- when no trapping of uncommon or exceptional cases is required
- when the VFP coprocessor is operating in RunFast mode
- when the hardware coprocessor is a VFPv3-based system.

For more information on using the VFP support code supplied with your installation of RVCT see *ARM Application Note 133 - Using VFP with RVDS*. You can find this application note in the `vfpsupport` subdirectory of the `Examples` directory of your RealView Development Suite distribution at *`install_directory`*`\RVDS\Examples\...\vfpsupport`.

5.6.4 Half-precision floating-point number support

Half-precision floating-point numbers are provided as an optional extension to the VFPv3 architecture. If the VFPv3 coprocessor is not available, or if a VFPv3 coprocessor is used that does not have this extension, they are supported through the floating-point library `fplib`.

Half-precision floating-point numbers can only be used when selected with the `fp16_format` command-line option. See *--fp16_format=format* on page 2-59 in the *Compiler Reference Guide*.

The half-precision floating-point formats available are `ieee` and `alternative`. In both formats, the basic layout of the 16-bit number is the same. See Figure 5-1 on page 5-36.

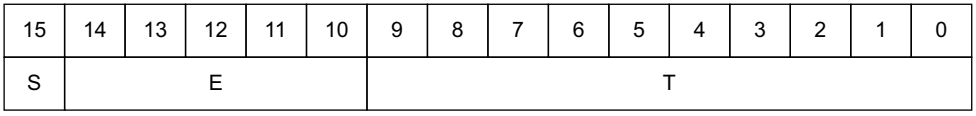

Figure 5-1 Half-precision floating-point format

Where:

```
S (bit[15]):      Sign bit
E (bits[14:10]):  Biased exponent
T (bits[9:0]):    Mantissa.
```

The meanings of these fields depend on the format that is selected.

IEEE half-precision

```
IF E==31:
   IF T==0: Value = Signed infinity
   IF T!=0: Value = Nan
             T[9] determines Quiet or Signalling:
                  0: Quiet NaN
                  1: Signalling NaN
IF 0<E<31:
   Value = (-1)^S x 2^(E-15) x (1+2^-10 T)

IF E==0:
   IF T==0: Value = Signed zero
   IF T!=0: Value = (-1)^S x 2^(-14) x (0+2^-10 T)
```

Alternative half-precision

```
IF 0<E<32:
   Value = (-1)^S x 2^(E-15) x (1+2^-10 T)

IF E==0:
   IF T==0: Value = Signed zero
   IF T!=0: Value = (-1)^S x 2^(-14) x (0+2^-10 T)
```

Usage restrictions

The following restrictions apply when you use the `__fp16` type:

- When used in a C or C++ expression, an `__fp16` type is promoted to single precision. Subsequent promotion to double precision can occur if required by one of the operands.

- A single precision value can be converted to `__fp16`. A double precision value is converted to single precision and then to `__fp16`, that could involve double rounding. This reflects the lack of direct double-to-16-bit conversion in the ARM architecture.
- When using `fpmode=fast`, no floating-point exceptions are raised when converting to and from half-precision floating-point format.
- Function formal arguments cannot be of type `__fp16`. However, pointers to variables of type `__fp16` can be used as function formal argument types.
- `__fp16` values can be passed as actual function arguments. In this case, they are converted to single-precision values.
- `__fp16` cannot be specified as the return type of a function. However, a pointer to an `__fp16` type can be used as a return type.
- An `__fp16` value is converted to a single-precision or double-precision value when used as a return value for a function that returns a `float` or `double`.

Name mangling

The C++ name mangling for the half-precision data type is specified in the C++ generic ABI. See the *C++ ABI for the ARM Architecture*.

5.6.5 Floating-point computations and linkage

It is important to understand the difference between floating-point computations and floating-point linkage. Floating-point computations are performed by hardware coprocessor instructions, or by library functions. Floating-point linkage is concerned with how arguments are passed between functions that use floating-point variables.

The types of floating-point linkage are:

- software floating-point linkage
- hardware floating-point linkage.

Software floating-point linkage means that the parameters and return value for a function are passed using the ARM integer registers `r0` to `r3` and the stack.

Hardware floating-point linkage uses the VFP coprocessor registers to pass the arguments and return value. For information on the VFP coprocessor registers, see the *ARM Procedure Call Standard for the ARM Architecture* (ARM IHI0042).

The benefit of using software floating-point linkage is that the resulting code can be run on a core with or without a VFP coprocessor. It is not dependent on the presence of a VFP hardware coprocessor, and it can be used with or without a VFP coprocessor present.

The benefit of using hardware floating-point linkage is that it is more efficient than software floating-point linkage, but you must have a VFP coprocessor.

Table 5-12 shows the compiler options available for the type of floating-point linkage and the type of floating-point computations you require.

Table 5-12 Compiler options for floating-point linkage and computations

Linkage		Computations		
Hardware floating-point linkage	**Software floating-point linkage**	**Hardware floating-point coprocessor**	**Software floating-point library (fplib)**	**Compiler options**
No	Yes	No	Yes	`--fpu=softvfp`

No	Yes	Yes	No	`--fpu=softvfp+vfpv2`
				`--fpu=softvfp+vfpv3`
				`--fpu=softvfp+vfpv3_fp16`
				`--fpu=softvfp+vfpv3_d16`
				`--fpu=softvfp+vfp3_d16_fp16`

Yes	No	Yes	No	`--fpu=vfp`
				`--fpu=vfpv2`
				`--fpu=vfpv3`
				`--fpu=vfpv3_fp16`
				`--fpu=vfpv3_dp16`
				`--fpu=vfpv3_d16_fp16`

`softvfp` specifies software floating-point linkage. When software floating-point linkage is used, either:

- the calling function and the called function must be compiled using one of the options `--softvfp`, `--fpu softvfp+vfpv2`, `--fpu softvfp+vfpv3`, `--fpu softvfp+vfpv3_fp16`, `softvfp+vfpv3_d16`, or `softvfp+vfpv3_d16_fp16`
- the calling function and the called function must be declared using the `__softfp` keyword.

Each of the options `--fpu softvfp`, `--fpu softvfp+vfpv2`,`--fpu softvfp+vfpv3`, `--fpu softvfp+vfpv3_fp16`, `--fpu softvfpv3_d16`, and `--fpu softvfpv3_d16_fp16` specify software floating-point linkage across the whole file. In contrast, the `__softfp` keyword enables software floating-point linkage to be specified on a function by function basis.

Note

Rather than having separate compiler options to select the type of floating-point linkage you require and the type of floating-point computations you require, you use one compiler option, `--fpu`, to select both. (See Table 5-12 on page 5-38.) For example, `--fpu=softvfp+vfpv2` selects *software* floating-point linkage, and a *hardware* coprocessor for the computations. Whenever you use `softvfp`, you are specifying software floating-point linkage.

See:

- *--fpu=name* on page 2-62 in the *Compiler Reference Guide*
- *__softfp* on page 4-15 in the *Compiler Reference Guide*
- *#pragma softfp_linkage, #pragma no_softfp_linkage* on page 4-70 in the *Compiler Reference Guide*.

5.7 Trapping and identifying division-by-zero errors

It is important to eliminate any division-by-zero errors in code, particularly for embedded systems that might not be able to recover easily. For ARM processor cores, division-by-zero errors fall into the following categories:

- integer division-by-zero errors
- (software) floating point division-by-zero errors.

Different techniques are required in both cases for trapping and identifying these errors.

5.7.1 Integer division

Integer division-by-zero errors can be trapped and identified by re-implementing the appropriate C library helper functions.

The default behavior when division by zero occurs is that when the signal function is used, or `__rt_raise` or `__aeabi_idiv0` are re-implemented, `__aeabi_idiv0` is called. Otherwise, the division function returns zero.

`__aeabi_idiv0` raises **SIGFPE** with an additional argument, `DIVBYZERO`.

Trapping division-by-zero errors in code

You can trap integer division-by-zero errors in the following ways:

- Re-implement the C library helper function `__aeabi_idiv0` so that division by zero returns some standard result, for example zero.

  Integer division is implemented in code through the C library helper functions `__aeabi_idiv` and `__aeabi_uidiv`. Both functions check for division by zero.

  When integer division by zero is detected, a branch to `__aeabi_idiv0` is made. To trap the division by zero, therefore, you have only to place a breakpoint on `__aeabi_idiv0`.

  See the *Run-time ABI for the ARM Architecture* for more information on the AEABI functions `__aeabi_idiv`, `__aeabi_uidiv`, and `__aeabi_idiv0`. This can be found at `http://www.arm.com/products/DevTools/ABI.html`.

- Re-implement the C library helper function `__rt_raise` to deal with the signal.

  By default, integer division by zero raises a signal. To intercept divide by zero, therefore, you can re-implement `__rt_raise`. This function has prototype:

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

  When a divide-by-zero error occurs, `__aeabi_idiv0` calls `__rt_raise(2, 2)`. Therefore, in your implementation of `__rt_raise`, you must check `(signal == 2) && (type == 2)` to determine if division by zero has occurred.

See:

— *Integer and floating point functions* on page 2-25 in the *Libraries Guide*

— *Exploiting the C library* on page 2-26 in the *Libraries Guide*

— *Tailoring error signaling, error handling, and program exit* on page 2-59 in the *Libraries Guide*

— *__rt_raise()* on page 2-62 in the *Libraries Guide*.

- Use the signal function to install a handler for `SIGFPE`. This method is more portable than the other methods described, but less efficient.

Identifying division-by-zero errors in code

On entry into `__aeabi_idiv0`, the link register `LR` contains the address of the instruction *after* the call to the `__aeabi_uidiv` division routine in your application code. To identify the offending line in your source code, you can look up the line of C code in the debugger at the address given by `LR`.

Examining parameters

If you want to examine parameters and save them for postmortem debugging, you can trap `__aeabi_idiv0`. You can intervene in all calls to `__aeabi_idiv0` by using the `$Super$$` and `$Sub$$` mechanism:

`$Super$$` Prefix `__aeabi_idiv0` with `$Super$$` to identify the original unpatched function `__aeabi_idiv0`. Use this to call the original function directly.

`$Sub$$` Prefix `__aeabi_idiv0` with `$Sub$$` to identify the new function to be called in place of the original version of `__aeabi_idiv0`. Use this to add processing before or after the original function `__aeabi_idiv0`.

Example 5-5 illustrates the use of the `$Super$$` and `$Sub$$` mechanism to intercept `__aeabi_div0`. See *Using $Super$$ and $Sub$$ to override symbol definitions* on page 4-18 in the *Linker User Guide*.

Example 5-5 Intercepting __aeabi_div0 using $Super$$ and $Sub$$

```
extern void $Super$$__aeabi_idiv0(void);
/* this function is called instead of the original __aeabi_idiv0() */
void $Sub$$__aeabi_idiv0()
{
    // insert code to process a divide by zero
    ...
```

```
    // call the original __aeabi_idiv0 function
    $Super$$__aeabi_idiv0();
}
```

5.7.2 (Software) Floating-point division

Floating-point division-by-zero errors in software can be trapped and identified using a combination of intrinsics and C library helper functions.

Trapping division-by-zero errors in code

To trap floating-point division-by-zero errors in your code, use the intrinsic:

```
__ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_DIVBYZERO);
```

This traps any division-by-zero errors in code, and untraps all other exceptions, as illustrated in Example 5-6.

Example 5-6 Division by zero error

```
#include <stdio.h>
#include <fenv.h>

int main(void)
{
    float a, b, c;
    // Trap the Invalid Operation exception and untrap all other exceptions:
    __ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_DIVBYZERO);
    c = 0;
    a = b / c;
    printf("b / c = %f, ", a); // trap division-by-zero error
    return 0;
}
```

Identifying division by zero errors in code

The C library helper function `_fp_trapveneer` is called whenever an exception occurs. On entry into this function, the state of the registers is unchanged from when the exception occurred. Therefore, to find the address of the function in your application code that contains the arithmetic operation that resulted in the exception, place a breakpoint on the function `_fp_trapveneer` and look at `LR`.

For example, suppose the C code of Example 5-6 is compiled from the command line using the string:

```
armcc --fpmode ieee_full
```

When the assembly language code produced by the compiler is disassembled, RealView Debugger produces the output shown in Example 5-7.

Example 5-7 Disassembly of division by zero error

```
main:
    00008080 E92D4010  PUSH    {r4,lr}
    00008084 E3A01C02  MOV     r1,#0x200
    00008088 E3A00C9F  MOV     r0,#0x9f00
    0000808C EB000F1A  BL      __ieee_status              <0xbcfc>
    00008090 E59F0020  LDR     r0,0x80b8
    00008094 E3A01000  MOV     r1,#0
    00008098 EB000DEA  BL      _fdiv                      <0xb848>
    0000809C EB000DBD  BL      _f2d                       <0xb798>
    000080A0 E1A02000  MOV     r2,r0
    000080A4 E1A03001  MOV     r3,r1
    000080A8 E28F000C  ADR     r0,{pc}+0x14 ; 0x80bc
    000080AC EB000006  BL      __0printf                  <0x80cc>
    000080B0 E3A00000  MOV     r0,#0
    000080B4 E8BD8010  POP     {r4,pc}
    000080B8 40A00000  <Data>  0x00 0x00 0xA0 '@'
    000080BC 202F2062  <Data>  'b' ' ' '/' ' '
    000080C0 203D2063  <Data>  'c' ' ' '=' ' '
    000080C4 202C6625  <Data>  '%' 'f' ',' ' '
    000080C8 00000000  <Data>  0x00 0x00 0x00 0x00
```

Placing a breakpoint on `_fp_trapveneer` and executing the disassembly in the debug monitor produces:

```
> go
Stopped at 0x0000BF6C due to SW Instruction BreakpointStopped at 0x0000BF6C:
TRAPV_S\_fp_trapveneer
```

Then, inspection of the registers shows:

```
  r0: 0x40A00000     r1: 0x00000000     r2: 0x00000000     r3: 0x00000000
  r4: 0x0000C1DC     r5: 0x0000C1CC     r6: 0x00000000     r7: 0x00000000
  r8: 0x00000000     r9: 0x00000000    r10: 0x0000C0D4    r11: 0x00000000
 r12: 0x08000004     SP: 0x07FFFFF8     LR: 0x0000809C     PC: 0x0000BF6C
CPSR: nzcvIFtSVC
```

The address contained in the link register `LR` is set to 0x809c, the address of the instruction after the instruction `BL _fdiv` that resulted in the exception.

Examining parameters

To save parameters for post-mortem debugging you must intercept `_fp_trapveneer`. To intervene in all calls to `_fp_trapveneer`, use the `$Super$$` and `$Sub$$` mechanism. For example:

```
    AREA foo, CODE
IMPORT |$Super$$_fp_trapveneer|
EXPORT |$Sub$$_fp_trapveneer|
    |$Sub$$_fp_trapveneer|
;; Add code to save whatever registers you require here
;; Take care not to corrupt any needed registers
    B |$Super$$_fp_trapveneer|
    END
```

See:

- *Integer division* on page 5-40
- *Using $Super$$ and $Sub$$ to override symbol definitions* on page 4-18 in the *Linker User Guide*.

5.8 New features of C99

The 1999 C standard introduces a range of new features into C, including:

- New language features, including new keywords and identifiers, together with extended syntax for the existing C90 language
- New library features, including new libraries, and new macros and functions for existing C90 libraries.

A selection of new features in C99 that are of interest to developers using them for the first time are described in the following sections.

Note

C90 is compatible with Standard C++ in the sense that the language specified by the standard is a subset of C++, except for a few special cases. New features in the C99 standard mean that C99 is no longer compatible with C++ in this sense.

5.8.1 Language features

The C99 standard introduces several new language features, including:

- Some features similar to extensions to C90 offered in the GNU compiler, for example, macros with a variable number of arguments.

  Note

  The implementations of extensions to C90 in the GNU compiler are not always compatible with the implementations of similar features in C99.

- Some features available in C++, such as // comments and the ability to mix declarations and code.
- Some entirely new features, for example complex numbers, restricted pointers and designated initializers.

A selection of new language features of C99 that might be of particular interest are described in the following sections.

// comments

You can use // to indicate the start of a one-line comment, like in C++. See *// comments* on page 3-5 in the *Compiler Reference Guide*.

Compound literals

ISO C99 supports compound literals. A compound literal looks like a cast followed by an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer. It is an lvalue. For example:

```
int y[] = (int []) {1, 2, 3};
int z[] = (int [3]) {1};
```

Designated initializers

In C90, there is no way to initialize specific members of arrays, structures, or unions. C99 supports the initialization of specific members of an array, structure, or union by either name or subscript through the use of *designated initializers*. For example:

```
typedef struct
{
    char *name;
    int rank;
} data;
data vars[10] = { [0].name = "foo", [0].rank = 1,
                  [1].name = "bar", [1].rank = 2,
                  [2].name = "baz",
                  [3].name = "gazonk" };
```

Members of an aggregate that are not explicitly initialized are initialized to zero by default.

Hex floats

C99 supports floating-point numbers that can be written in hexadecimal format. For example:

```
float hex_floats(void)
{
    return 0x1.fp3; // 1 15/16 * 2^3
}
```

In hexadecimal format the exponent is a decimal number that indicates the power of two by which the significant part is multiplied. Therefore `0x1.fp3 = 1.9375 * 8 = 1.55e1`.

Flexible array members

In a **`struct`** with more than one member, the last member of the **`struct`** can have incomplete array type. Such a member is called a *flexible array member* of the **`struct`**.

Note

When a **struct** has a flexible array member, the entire **struct** itself has incomplete type.

Flexible array members enable you to mimic dynamic type specification in C in the sense that you can defer the specification of the array size to runtime. For example:

```
extern const int n;
typedef struct
{
    int len;
    char p[];
} str;
void foo(void){
    size_t str_size = sizeof(str);  // equivalent to offsetoff(str, p)
    str *s = malloc(str_size + (sizeof(char) * n));
}
```

__func__ predefined identifier

The __func__ predefined identifier provides a means of obtaining the name of the current function. For example, the function:

```
void foo(void)
{
    printf("This function is called '%s'.\n", __func__);
}
```

prints:

```
This function is called 'foo'.
```

inline functions

The C99 keyword **inline** hints to the compiler that invocations of a function qualified with **inline** are to be expanded inline. For example:

```
inline int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

The compiler inlines a function qualified with **inline** only if it is reasonable to do so. It is free to ignore the hint if inlining the function adversely affects performance. See *Function inlining* on page 5-18.

Note

The semantics of **inline** in C99 are different to the semantics of **inline** in Standard C++.

long long data type

C99 supports the integral data type **long long**. This type is 64 bits wide in RVCT. For example:

```
long long int j = 25902068371200;                // length of light day, meters
unsigned long long int i = 946073047258080000ULL; // length of light year, meters
```

See *long long* on page 3-8 in the *Compiler Reference Guide*.

Macros with a variable number of arguments

You can declare a macro in C99 that accepts a variable number of arguments. The syntax for defining such a macro is similar to that of a function. For example:

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
void Variadic_Macros_0()
{
    debug ("a test string is printed out along with %x %x %x\n", 12, 14, 20);
}
```

Mixed declarations and code

C99 enables you to mix declarations and code within compound statements, like in C++. For example:

```
void foo(float i)
{
    i = (i > 0) ? -i : i;
    float j = sqrt(i);    // illegal in C90
}
```

New block scopes for selection and iteration statements

In a **for** loop, the first expression can be a declaration, like in C++. The scope of the declaration extends to the body of the loop only. For example:

```
extern int max;
for (int n = max - 1; n >= 0; n--)
{
    // body of loop
}
```

is equivalent to:

```
extern int max;
{
    int n = max - 1;
    for (; n >= 0; n--)
    {
        // body of loop    }
}
```

Note

Unlike in C++, you cannot introduce new declarations in a **for**-test, **if**-test or **switch**-expression.

_Pragma preprocessing operator

C90 does not permit a #pragma directive to be produced as the result of a macro expansion. The C99 _Pragma operator enables you to embed a preprocessor macro in a pragma directive. For example:

```
# define RWDATA(X) PRAGMA(arm section rwdata=#X)
# define PRAGMA(X) _Pragma(#X)
RWDATA(foo)  // same as #pragma arm section rwdata="foo"
int y = 1;   // y is placed in section "foo"
```

Restricted pointers

The C99 keyword **restrict** enables you to ensure that different object pointer types and function parameter arrays do not point to overlapping regions of memory. This enables the compiler to perform optimizations that might otherwise be prevented because of possible aliasing.

In the following example, pointer a does not, and cannot, point to the same region of memory as pointer b:

```
void copy_array(int n, int *restrict a, int *restrict b)
{
    while (n-- > 0)
        *a++ = *b++;
}
void test(void)
{
    extern int array[100];
    copy_array(50, array + 50, array);    // valid
    copy_array(50, array + 1, array);     // undefined behavior
}
```

Pointers qualified with **restrict** can however point to different arrays, or to different regions within an array.

5.8.2 Library features

The C99 standard introduces several new library features of interest to programmers, including:

- Some features similar to extensions to the C90 standard libraries offered in UNIX standard libraries, for example, the `snprintf` family of functions.
- Some entirely new library features, for example, the standardized floating-point environment offered in `<fenv.h>`.

A selection of new library features of C99 that might be of particular interest are described in the following sections.

Additional math library functions in `<math.h>`

C99 supports additional macros, types, and functions in the standard header `<math.h>` that are not found in the corresponding C90 standard header.

New macros found in C99 that are not found in C90 include:

```
INFINITY // positive infinity
NAN      // IEEE not-a-number
```

New generic function macros found in C99 that are not found in C90 include:

```
#define isinf(x) // non-zero only if x is positive or negative infinity
#define isnan(x) // non-zero only if x is NaN
#define isless(x, y) // 1 only if x < y and x and y are not NaN, and 0 otherwise
#define isunordered(x, y) // 1 only if either x or y is NaN, and 0 otherwise
```

New mathematical functions found in C99 that are not found in C90 include:

```
double acosh(double x); // hyperbolic arccosine of x
double asinh(double x); // hyperbolic arcsine of x
double atanh(double x); // hyperbolic arctangent of x
double erf(double x); // returns the error function of x
double round(double x); // returns x rounded to the nearest integer
double tgamma(double x); // returns the gamma function of x
```

C99 supports the new mathematical functions for all real floating-point types.

Single precision versions of all existing `<math.h>` functions are also supported.

Complex numbers

In C99 mode, the compiler supports complex and imaginary numbers. In GNU mode, the compiler supports complex numbers only.

For example:

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    complex float z = 64.0 + 64.0*I;
    printf("z = %f + %fI\n", creal(z), cimag(z));
    return 0;
}
```

The complex types are:

- **`float complex`**
- **`double complex`**
- **`long double complex`**.

Boolean type and `<stdbool.h>`

C99 introduces the native type **`_Bool`**. The associated standard header `<stdbool.h>` introduces the macros `bool`, `true` and `false` for Boolean tests. For example:

```
#include <stdbool.h>
bool foo(FILE *str)
{
    bool err = false;
    ...
    if (!fflush(str))
    {
        err = true;
    }
    ...
    return err;
}
```

Note

The C99 semantics for bool are intended to match those of C++.

Extended integer types and functions in <inttypes.h> and <stdint.h>

In C90, the **long** data type can serve both as the largest integral type, and as a 32-bit container. C99 removes this ambiguity through the new standard library header files <inttypes.h> and <stdint.h>.

The header file <stdint.h> introduces the new types:

- intmax_t and uintmax_t, that are maximum width signed and unsigned integer types
- intptr_t and unintptr_t, that are integer types capable of holding signed and unsigned object pointers.

The header file <inttypes.h> provides library functions for manipulating values of type intmax_t, including:

```
intmax_t imaxabs(intmax_t x); // absolute value of x
imaxdiv_t imaxdiv(intmax_t x, intmax_t y) // returns the quotient and remainder
                                          // of x / y
```

Floating-point environment access in <fenv.h>

The C99 standard header file <fenv.h> provides access to an IEEE 754-compliant floating-point environment for numerical programming. The library introduces two types and numerous macros and functions for managing and controlling floating-point state.

The new types supported are:

- fenv_t, representing the entire floating-point environment
- fexcept_t, representing the floating-point state.

New macros supported include:

- FE_DIVBYZERO, FE_INEXACT, FE_INVALID, FE_OVERFLOW and FE_UNDERFLOW for managing floating-point exceptions
- FE_DOWNWARD, FE_TONEAREST, FE_TOWARDZERO, FE_UPWARD for managing rounding in the represented rounding direction
- FE_DFL_ENV, representing the default floating-point environment.

New functions include:

```
int feclearexcept(int ex); // clear floating-point exceptions selected by ex
int feraiseexcept(int ex); // raise floating point exceptions selected by ex
int fetestexcept(int ex); // test floating point exceptions selected by x
int fegetround(void); // return the current rounding mode
```

```
int fesetround(int mode); // set the current rounding mode given by mode
int fegetenv(fenv_t *penv); return the floating-point environment in penv
int fesetenv(const fenv_t *penv); // set the floating-point environment to penv
```

snprintf family of functions in <stdio.h>

Using the sprintf family of functions found in the C90 standard header <stdio.h> can be dangerous. In the statement:

```
sprintf(buffer, size, "Error %d: Cannot open file '%s'", errno, filename);
```

the variable size specifies the minimum number of characters to be inserted into buffer. Consequently, more characters can be output than might fit in the memory allocated to the string.

The snprintf functions found in the C99 version of <stdio.h> are safe versions of the sprintf functions that prevent buffer overrun. In the statement:

```
snprintf(buffer, size, "Error %d: Cannot open file '%s'", errno, filename);
```

the variable size specifies the maximum number of characters that can be inserted into buffer. The buffer can never be overrun, provided its size is always greater than the size specified by size.

Type-generic math macros in <tgmath.h>

The new standard header <tgmath.h> defines several families of mathematical functions that are type generic in the sense that they are overloaded on floating-point types. For example, the trigonometric function cos works as if it has the overloaded declaration:

```
extern float cos(float x);
extern double cos(double x);
extern long double cos(long double x);
...
```

A statement such as:

```
p = cos(0.78539f); // p = cos(pi / 4)
```

calls the single-precision version of the cos function, as determined by the type of the literal 0.78539f.

Note

Type-generic families of mathematical functions can be defined in C++ using the operator overloading mechanism. The semantics of type-generic families of functions defined using operator overloading in C++ are different from the semantics of the corresponding families of type-generic functions defined in <tgmath.h>.

Wide character I/O functions in <wchar.h>

Wide character I/O functions have been introduced in C99. These enable you to read and write wide characters from a file in much the same way as normal characters. The ARM C Library supports all of the C99 functions defined in wchar.h. See section 7.24 of *ISO/IEC 9899:TC2*.

Chapter 6
Diagnostic Messages

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

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

This section includes the following subsections:

6.1 Redirecting diagnostics

Use the `--errors=filename` option to redirect compiler diagnostic output to a file. Diagnostics that relate to the command options are not redirected.

See *Controlling the output of diagnostic messages* on page 6-4.

6.2 Severity of diagnostic messages

Diagnostic messages have an associated *severity*, as described in Table 6-1.

Table 6-1 Severity of diagnostic messages

Severity	Description
Internal fault	Internal faults indicate an internal problem with the compiler. Contact your supplier with the information listed in Feedback on RealView Compilation Tools on page xiii.
Error	Errors indicate problems that cause the compilation to stop. These errors include command line errors, internal errors, missing include files, and violations in the syntactic or semantic rules of the C or C++ language. If multiple source files are specified, then no further source files are compiled.
Warning	Warnings indicate unusual conditions in your code that might indicate a problem. Compilation continues, and object code is generated unless any further problems with an Error severity are detected.
Remark	Remarks indicate common, but sometimes unconventional, use of C or C++. These diagnostics are not displayed by default. Compilation continues, and object code is generated unless any further problems with an Error severity are detected.

6.3 Controlling the output of diagnostic messages

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

`--no_brief_diagnostics, --brief_diagnostics`

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

`--diag_style={arm|ide|gnu}`

Specifies the style used to display diagnostic messages.

`--errors=`*`filename`*

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

`--remarks` Causes the compiler to issue remark messages, such as warning of padding in structures. Remarks are not issued by default.

`--no_wrap_diagnostics, --wrap_diagnostics`

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

See *Command-line options* on page 2-2 in the *Compiler Reference Guide*.

6.4 Changing the severity of diagnostic messages

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

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

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

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

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

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

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

These options require a comma-separated list of the error messages that you want to change. For example, you might want to change a warning message with the number #1293 to Remark severity, because remarks are not displayed by default.

To do this, use the following command:

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

Note

These options also have pragma equivalents. See *Pragmas* on page 4-14.

The following diagnostic messages can be changed:

- Messages with the number format `#`*`nnnn`*`-D`.
- Warning messages with the number format `C`*`nnnn`*`W`.

6.5 Suppressing diagnostic messages

To suppress all diagnostic messages that have the specified tag(s) use the following option: --diag_suppress=*tag*[, *tag*, ...]

See also:

- *Pragmas* on page 4-14
- *--diag_suppress=tag[,tag,...]* on page 2-47 in the *Compiler Reference Guide*.

6.6 Prefix letters in diagnostic messages

The RVCT tools automatically insert an identification letter to diagnostic messages, as described in Table 6-2. Using these prefix letters enables the RVCT tools to use overlapping message ranges.

Table 6-2 Identifying diagnostic messages

Prefix letter	RVCT tool
C	armcc
A	armasm
L	armlink or armar
Q	fromelf

The following rules apply:

- All the RVCT tools act on a message number without a prefix.
- A message number with a prefix is only acted on by the tool with the matching prefix.
- A tool does not act on a message with a non-matching prefix.

Thus, the compiler prefix `C` can be used with `--diag_error`, `--diag_remark`, and `--diag_warning`, or when suppressing messages, for example:

```
armcc --diag_suppress=C1287,C3017 ...
```

Use the prefix letters to control options that are passed from the compiler to other tools, for example, include the prefix letter `L` to specify linker message numbers.

6.7 Suppressing warning messages with -W

The `-W` option suppresses all warnings.

6.8 Exit status codes and termination messages

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

filename: *n* warnings, *n* errors

where *n* indicates the number of warnings or errors detected.

Note

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

This section also includes:

- *Response to signals*
- *Exit status*.

6.8.1 Response to signals

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

6.8.2 Exit status

On completion, the compiler returns a value greater than zero if an error is detected. If no error is detected, a value of zero is returned.

See *Severity of diagnostic messages* on page 6-3 for more information on how the compiler handles the different levels of diagnostic messages.

6.9 Data flow warnings

The compiler performs data flow analysis as part of its optimization process. This information can be used to identify potential problems in your code, for example, to issue warnings about the use of uninitialized variables.

The data flow analysis can only warn about local variables that are held in processor registers, not global variables held in memory or variables or structures that are placed on the stack.

Be aware that:

- Data flow warnings are issued by default (in RVCT v2.0 and earlier, data flow warnings are issued only if the `-fa` option is specified).
- Data flow analysis is disabled at `-O0` (even if the `-fa` option is specified).

The results of this analysis vary with the level of optimization used. This means that higher optimization levels might produce a number of warnings that do not appear at lower levels. For example, the following source code results in the compiler generating the warning `C3017W: i may be used before being set`, at `-O2`:

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

The data flow analysis cannot reliably identify faulty code and any `C3017W` warnings issued by the compiler are intended only as an indication of possible problems. For a full analysis of your code, suppress this warning with `--diag_suppress=C3017` and then use any appropriate third-party analysis tool, for example Lint.

Chapter 7
Using the Inline and Embedded Assemblers

This chapter describes the optimizing inline assembler and non-optimizing embedded assembler of the ARM compiler, `armcc`. It contains the following sections:

7.1 Inline assembler

The ARM compiler provides an inline assembler that enables you to write optimized assembly language routines, and access features of the target processor not available from C or C++.

The following subsections are included:

- *Inline assembler support*
- *Inline assembler syntax* on page 7-3
- *Restrictions on inline assembly operations* on page 7-5
- *Virtual registers* on page 7-8
- *Constants* on page 7-9
- *Instruction expansion* on page 7-9
- *Condition flags* on page 7-10
- *Operands* on page 7-10
- *Function calls and branches* on page 7-12
- *Labels* on page 7-15
- *Differences from previous versions of the ARM C/C++ compilers* on page 7-15.

See also:

- Chapter 4 *Mixing C, C++, and Assembly Language* in the *Developer Guide* for information on how to use the inline assembler in C and C++ source code, and restrictions on inline assembly language
- the *Assembler Guide* for more information on writing assembly language for the ARM processors.

7.1.1 Inline assembler support

The inline assembler supports ARM assembly language only. It does not support:

- Thumb assembly language
- Thumb-2 assembly language
- ARMv7 instructions
- VFP instructions
- NEON instructions.

You can use the embedded assembler for Thumb and Thumb-2 support.

Most ARMv6 instructions are supported by the inline assembler, including the complete set of ARMv6 SIMD instructions. ARMv6 instructions that the inline assembler does not support are `SETEND` and some of the system extensions.

Most ARMv5 instructions are supported by the inline assembler, including generic coprocessor instructions. ARMv5 instructions that the inline assembler does not support are `BX`, `BLX`, and `BXJ`.

7.1.2 Inline assembler syntax

The ARM compiler supports an extended inline assembler syntax, introduced by the **`asm`** keyword (C++), or the **`__asm`** keyword (C and C++). The syntax for these keywords is described in the following sections:

- *Inline assembly with the __asm keyword*
- *Inline assembly with the asm keyword*
- *Rules for using __asm and asm* on page 7-4.

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

Inline assembly with the __asm keyword

The inline assembler is invoked with the assembler specifier, and is followed by a list of assembler instructions inside braces or parentheses. You can specify inline assembler code using the following formats:

- On a single line, for example:

  ```
  __asm("instruction[;instruction]"); // Must be a single string
  __asm{instruction[;instruction]}
  ```

  You cannot include comments.

- On multiple lines, for example:

  ```
  __asm
  {
      ...
      instruction
      ...
  }
  ```

  You can use C or C++ comments anywhere in an inline assembly language block.

Also, see *Rules for using __asm and asm* on page 7-4.

Inline assembly with the asm keyword

When compiling C++, the ARM compiler supports the **`asm`** syntax proposed in the ISO C++ Standard. You can specify inline assembler code using the following formats:

- On a single line, for example:

```
asm("instruction[;instruction]"); // Must be a single string
asm{instruction[;instruction]}
```

You cannot include comments.

- On multiple lines, for example:

```
asm
{
    ...
    instruction
    ...
}
```

You can use C or C++ comments anywhere in an inline assembly language block.

Rules for using __asm and asm

Follow these rules when using the **__asm** and **asm** keywords:

- If you include multiple instructions on the same line, you must separate them with a semicolon (;). If you use double quotes, you must enclose all the instructions within a single set of double quotes (").
- If an instruction requires more than one line, you must specify the line continuation with the backslash character (\).
- For the multiple line format, you can use C or C++ comments anywhere in the inline assembly language block. However, you cannot embed comments in a line that contains multiple instructions.
- The comma (,) is used as a separator in assembly language, so C expressions with the comma operator must be enclosed in parentheses to distinguish them:

```
__asm
{
    ADD x, y, (f(), z)
}
```

- An **asm** statement must be inside a C++ function. An **asm** statement can be used anywhere a C++ statement is expected.
- Register names in the inline assembler are treated as C or C++ variables. They do not necessarily relate to the physical register of the same name (see *Virtual registers* on page 7-8). If you do not declare the register as a C or C++ variable, the compiler generates a warning.
- Do not save and restore registers in inline assembler. The compiler does this for you. Also, the inline assembler does not provide direct access to the physical registers. See *Virtual registers* on page 7-8.

If registers other than CPSR and SPSR are read without being written to, an error message is issued. For example:

```
int f(int x)
{
    __asm
    {
        STMFD sp!, {r0}    // save r0 - illegal: read before write
        ADD r0, x, 1
        EOR x, r0, x
        LDMFD sp!, {r0}    // restore r0 - not needed.
    }
    return x;
}
```

The function must be written as:

```
int f(int x)
{
    int r0;
    __asm
    {
        ADD r0, x, 1
        EOR x, r0, x
    }
    return x;
}
```

See *Restrictions on inline assembly operations*.

7.1.3 Restrictions on inline assembly operations

There are a number of restrictions on the operations that can be performed in inline assembly code. These restrictions provide a measure of safety, and ensure that the assumptions in compiled C and C++ code are not violated in the assembled assembly code.

Miscellaneous restrictions

The inline assembler has the following restrictions:

- The inline assembler is a high-level assembler, and the code it generates might not always be exactly what you write. Do not use it to generate more efficient code than the compiler generates. Use embedded assembler or the ARM assembler armasm for this purpose.
- Some low-level features that are available in the ARM assembler armasm, such as branching and writing to PC, are not supported.

- Label expressions are not supported.
- You cannot get the address of the current instruction using dot notation (.) or `{PC}`.
- The `&` operator cannot be used to denote hexadecimal constants. Use the `0x` prefix instead. For example:

  ```
  __asm { AND x, y, 0xF00 }
  ```

- The notation to specify the actual rotation of an 8-bit constant is not available in inline assembly language. This means that where an 8-bit shifted constant is used, the C flag must be regarded as corrupted if the NZCV flags are updated.
- You must not modify the stack. This is not necessary because the compiler automatically stacks and restores any working registers as required. The compiler does not permit you to explicitly stack and restore work registers.

Registers

Registers such as `r0-r3`, `sp`, `lr`, and the NZCV flags in the `CPSR` must be used with caution. If you use C or C++ expressions, these might be used as temporary registers and NZCV flags might be corrupted by the compiler when evaluating the expression. See *Virtual registers* on page 7-8.

The `pc`, `lr`, and `sp` registers cannot be explicitly read or modified using inline assembly code because there is no direct access to any physical registers. However, you can use the following intrinsics described in the *Compiler Reference Guide* to access these registers:

- *__current_pc* on page 4-78
- *__current_sp* on page 4-78
- *__return_address* on page 4-95.

Processor modes

You can change processor modes or modify coprocessor states, but the compiler does not recognize these changes. If you change processor mode, you must not use C or C++ expressions until you change back to the original mode, otherwise the compiler corrupts the registers for the new processor mode.

Similarly, if you change the state of a floating-point coprocessor by executing floating-point instructions, you must not use floating-point expressions until the original state has been restored.

Thumb instruction set

The inline assembler is not available when compiling C or C++ for Thumb state, and the inline assembler does not assemble Thumb instructions. Instead, the compiler switches to ARM state automatically.

If you want to include inline assembly in a source file that contains code to be compiled for Thumb, enclose the functions containing inline assembler code between `#pragma arm` and `#pragma thumb` statements. For example:

```
...         // Thumb code
#pragma arm // ARM code. Switch code generation to the ARM instruction set so
            // that the inline assembler is available.

int add(int i, int j)
{
    int res;
    __asm
    {
        ADD   res, i, j   // add here
    }
    return res;
}
#pragma thumb   // Thumb code. Switch back to the Thumb instruction set.
                // The inline assembler is no longer available.
```

You must also compile your code using the `--apcs /interwork` compiler option.

See:

- *Interworking qualifiers* on page 2-24
- *Pragmas* on page 4-58 in the *Compiler Reference Guide*.

VFP coprocessor

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

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

Unsupported instructions

The following instructions are not supported in the inline assembler:

- `BKPT`, `BX`, `BXJ`, and `BLX` instructions

Note

You can insert a `BKPT` instruction in C and C++ code by using the `__breakpoint()` intrinsic.

- `LDR Rn, =expression` pseudo-instruction. Use `MOV Rn, expression` instead. (This can generate a load from a literal pool.)
- `LDRT`, `LDRBT`, `STRT`, and `STRBT` instructions
- `MUL`, `MLA`, `UMULL`, `UMLAL`, `SMULL`, and `SMLAL` flag setting instructions
- `MOV` or `MVN` flag-setting instructions where the second operand is a constant
- user-mode `LDM` instructions
- `ADR` and `ADRL` pseudo-instructions.

See *__breakpoint* on page 4-75 in the *Compiler Reference Guide*.

7.1.4 Virtual registers

The inline assembler provides no direct access to the physical registers of an ARM processor. If an ARM register name is used as an operand in an inline assembler instruction it becomes a reference to a virtual register, with the same name, and not the physical ARM register.

The compiler allocates physical registers to each virtual register as appropriate during optimization and code generation. However, the physical register used in the assembled code might be different to that specified in the instruction. You can explicitly define these virtual registers as normal C or C++ variables. If they are not defined then the compiler supplies implicit definitions for the virtual registers.

The compiler-defined virtual registers have function local scope, that is, within a single function, multiple **`asm`** statements or declarations that reference the same virtual register name access the same virtual register.

No virtual registers are created for the `sp` (`r13`), `lr` (`r14`), and `pc` (`r15`) registers, and they cannot be read or directly modified in inline assembly code. See *Legacy inline assembler that accesses sp, lr, or pc* on page 7-27 for information on how you can modify your source code.

There is no virtual *Processor Status Register* (PSR). Any references to the `PSR` are always to the physical `PSR`.

Existing inline assembler code that conforms to previously documented guidelines continues to perform the same function as in previous versions of the compiler, although the actual registers used in each instruction might be different.

The initial value in each virtual register is unpredictable. You must write to virtual registers before reading them. The compiler generates an error if you attempt to read a virtual register before writing to it, for example, if you attempt to read the virtual register associated with the variable `r1`.

You must also explicitly declare the names of the variables in your C or C++ code. It is better to use C or C++ variables as instruction operands. The compiler generates a warning the first time a virtual or physical register name is used, and only once for each translation unit. For example, if you specify register `r3`, a warning is displayed.

7.1.5 Constants

The constant expression specifier `#` is optional. If it is used, the expression following it must be a constant.

7.1.6 Instruction expansion

An ARM instruction in inline assembly code might be expanded into several instructions in the compiled object. The expansion depends on the instruction, the number of operands specified in the instruction, and the type and value of each operand.

Instructions using constants

The constant in an instruction with a constant operand is not limited to the values permitted by the instruction. Instead, the compiler translates the instruction into a sequence of instructions with the same effect. For example:

```
ADD r0,r0,#1023
```

might be translated into:

```
ADD r0,r0,#1024
SUB r0,r0,#1
```

With the exception of coprocessor instructions, all ARM instructions with a constant operand support instruction expansion. In addition, the `MUL` instruction can be expanded into a sequence of adds and shifts when the third operand is a constant.

The effect of updating the `CPSR` by an expanded instruction is:

- arithmetic instructions set the NZCV flags correctly

- logical instructions:
  - set the NZ flags correctly
  - do not change the V flag
  - corrupt the C flag.

Load and store instructions

The LDM, STM, LDRD, and STRD instructions might be replaced by equivalent ARM instructions. In this case the compiler outputs a warning message informing you that it might expand instructions.

Inline assembly code must be written in such a way that it does not depend on the number of expected instructions or on the expected execution time for each specified instruction.

Instructions that normally place constraints on pairs of operand registers, such as LDRD and STRD, are replaced by a sequence of instructions with equivalent functionality and without the constraints. However, these might be recombined into LDRD and STRD instructions.

All LDM and STM instructions are expanded into a sequence of LDR and STR instructions with equivalent effect. However, the compiler might subsequently recombine the separate instructions into an LDM or STM during optimization.

7.1.7 Condition flags

An inline assembly instruction might explicitly or implicitly attempt to update the processor condition flags. Inline assembly instructions that involve only virtual register operands or simple expression operands (see *Operands*) have predictable behavior. The condition flags are set by the instruction if either an implicit or an explicit update is specified. The condition flags are unchanged if no update is specified. If any of the instruction operands are not simple operands, then the condition flags might be corrupted unless the instruction updates them. In general, the compiler cannot easily diagnose potential corruption of the condition flags. However, for operands that require the construction and subsequent destruction of C++ temporaries the compiler gives a warning if the instruction attempts to update the condition flags. This is because the destruction might corrupt the condition flags.

7.1.8 Operands

Operands can be one of several types:

Virtual registers

Registers specified in inline assembly instructions always denote virtual registers and not the physical ARM integer registers. Virtual registers require no declaration, and the size of the virtual registers is the same as the physical registers. However, the physical register used in the assembled code might be different to that specified in the instruction. See *Virtual registers* on page 7-8.

Expression operands

Function arguments, C or C++ variables, and other C or C++ expressions can be specified as register operands in an inline assembly instruction.

The type of an expression used in place of an ARM integer register must be either an integral type (that is, **`char`**, **`short`**, **`int`** or **`long`**), excluding **`long long`**, or a pointer type. No sign extension is performed on **`char`** or **`short`** types. You must perform sign extension explicitly for these types. The compiler might add code to evaluate these expressions and allocate them to registers.

When an operand is used as a destination, the expression must be a modifiable lvalue if used as an operand where the register is modified. For example, a destination register or a base register with a base-register update.

For an instruction containing more than one expression operand, the order that expression operands are evaluated is unspecified.

An expression operand of a conditional instruction is only evaluated if the conditions for the instruction are met.

A C or C++ expression that is used as an inline assembler operand might result in the instruction being expanded into several instructions. This happens if the value of the expression does not meet the constraints set out for the instruction operands in the *ARM Architecture Reference Manual*.

If an expression used as an operand creates a temporary that requires destruction, then the destruction occurs after the inline assembly instruction is executed. This is analogous to the C++ rules for destruction of temporaries.

A simple expression operand is one of the following:

- a variable value
- the address of a variable
- the dereferencing of a pointer variable
- a compile-time constant.

Any expression containing one of the following is not a simple expression operand:

- an implicit function call, such as for division, or explicit function call
- the construction of a C++ temporary
- an arithmetic or logical operation.

Register lists

A register list can contain a maximum of 16 operands. These operands can be virtual registers or expression register operands.

The order that virtual registers and expression operands are specified in a register list is significant. The register list operands are read or written in left-to-right order. The first operand uses the lowest address, and subsequent operands use addresses formed by incrementing the previous address by four. This new behavior is in contrast to the usual operation of the `LDM` or `STM` instructions where the lowest numbered physical register is always stored to the lowest memory address. This difference in behavior is a consequence of the virtualization of registers.

An expression operand or virtual register can appear more than once in a register list and is used each time it is specified.

The base register is updated, if specified. The update overwrites any value loaded into the base register during a memory load operation.

Operating on User mode registers when in a privileged mode, by specifying `^` after a register list, is not supported by the inline assembler.

Intermediate operands

A C or C++ constant expression of an integral type might be used as an immediate value in an inline assembly instruction.

A constant expression that is used to specify an immediate shift must have a value that lies in the range defined in the *ARM Architecture Reference Manual*, as appropriate for the shift operation.

A constant expression that is used to specify an immediate offset for a memory or coprocessor data transfer instruction must have a value with suitable alignment.

7.1.9 Function calls and branches

The `BL` and `SVC` instructions of the inline assembler enable you to specify three optional lists following the normal instruction fields. These instructions have the following format:

```
SVC{cond} svc_num, {input_param_list}, {output_value_list}, {corrupt_reg_list}
BL{cond} function, {input_param_list}, {output_value_list}, {corrupt_reg_list}
```

Note

The SVC instruction used to be named SWI. The inline assembler still accepts SWI in place of SVC.

The lists are described in the following sections:

- *No lists specified*
- *Input parameter list*
- *Output value list* on page 7-14
- *Corrupted register list* on page 7-14.

Note

- The BX, BLX, and BXJ instructions are not supported in the inline assembler.
- It is not possible to specify the lr, sp, or pc registers in any of the input, output, or corrupted register lists.
- The sp register must not be changed by any SVC instruction or function call.

No lists specified

If you do not specify any lists, then:

- r0-r3 are used as input parameters
- r0 is used for the output value
- r12 and r14 can be corrupted.

Input parameter list

This list specifies the expressions or variables that are the input parameters to the function call or SVC instruction, and the physical registers that contain the expressions or variables. They are specified as assignments to physical registers or as physical register names. A single list can contain both types of input register specification.

The inline assembler ensures that the correct values are present in the specified physical registers before the BL or SVC instruction is entered. A physical register name that is specified without assignment ensures that the value in the virtual register of the same name is present in the physical register. This ensures backwards compatibility with existing inline assembler code.

For example, the instruction:

```
BL foo, { r0=expression1, r1=expression2, r2 }
```

generates the following pseudocode:

```
MOV (physical) r0, expression1
MOV (physical) r1, expression2
MOV (physical) r2, (virtual) r2
BL foo
```

Output value list

This list specifies the physical registers that contain the output values from the BL or SVC and where they must be stored. The output values are specified as assignments from physical registers to modifiable lvalue expressions or as single physical register names.

The inline assembler takes the values from the specified physical registers and assigns them into the specified expressions. A physical register name specified without assignment causes the virtual register of the same name to be updated with the value from the physical register.

For example, the instruction:

```
BL foo, { }, { result1=r0, r1 }
```

generates the following pseudocode:

```
BL foo
MOV result1, (physical) r0
MOV (virtual) r1, (physical) r1
```

Corrupted register list

This list specifies the physical registers that are corrupted by the called function. If the condition flags are modified by the called function then you must specify the PSR in the corrupted register list.

The BL and SVC instructions always corrupt lr.

If this list is omitted then for BL and SVC, the registers r0-r3, pc, lr and the PSR are corrupted.

The branch instruction, B, must only be used to jump to labels within a single C or C++ function.

7.1.10 Labels

Labels defined in inline assembly code can be used as targets for branches or C and C++ `goto` statements. Labels defined in C and C++ can be used as targets by branch instructions in inline assembly code, in the form:

`BL{`*`cond`*`} label`

7.1.11 Differences from previous versions of the ARM C/C++ compilers

There are significant differences between the inline assembler in the ARM compiler and the inline assembler in previous versions of the ARM C and C++ compilers. This section highlights the main differences. For more information on using existing assembly code for the inline assembler see the *RealView Compilation Tools Developer Guide*.

ARMv6 instructions

Of all the ARMv6 instructions, the inline assembler supports the ARMv6 media instructions only.

Virtual registers

Inline assembly code for the compiler always specifies virtual registers. The compiler chooses the physical registers to be used for each instruction during code-generation, and enables the compiler to optimize fully the assembly code and surrounding C or C++ code.

The `pc` (`r15`), `lr` (`r14`), and `sp` (`r13`) registers cannot be accessed at all. An error message is generated when these registers are accessed.

The initial values of virtual registers are undefined. Therefore, you must write to virtual registers before reading them. The compiler warns you if code reads a virtual register before writing to it. The compiler also generates these warnings for legacy code that relies on particular values in physical registers at the beginning of inline assembly code, for example:

```
int add(int i, int j)
{
    int res;
    __asm
    {
        ADD res, r0, r1   // relies on i passed in r0 and j passed in r1
    }
    return res;
}
```

This code generates warning and error messages.

The errors are generated because virtual registers `r0` and `r1` are read before writing to them. The warnings are generated because `r0` and `r1` must be defined as C or C++ variables. The corrected code is:

```
int add(int i, int j)
{
    int res;
    __asm
    {
        ADD res, i, j
    }
    return res;
}
```

Instruction expansion

The inline assembler in the compiler expands the instructions `LDM`, `STM`, `LDRD`, and `STRD` into a sequence of single-register memory operations that perform the equivalent functionality.

It is possible that the compiler optimizes the sequence of single-register memory operation instructions back into a multiple-register memory operation.

Register lists

The order of operands in a register list for an `LDM` or `STM` instruction is significant. They are used in the order given, that is left-to-right, and the first operand references the lowest generated memory address. This is in contrast to the behavior in previous compilers where the lowest numbered register always referenced the lowest memory address generated by the instruction.

This has changed because you can now use expression operands in register lists alongside virtual registers. The compiler gives a warning message if it encounters a register list that contains only virtual registers, and where the result of the new ordering is different to that from previous ARM C and C++ compilers.

Thumb instructions

The inline assembler in the compiler does not support the Thumb® instruction set. It does not assemble Thumb instructions, and cannot be used at all when compiling C or C++ for Thumb state, unless you use the `#pragma arm` and `#pragma thumb` pragmas (see *Thumb instruction set* on page 7-7).

7.2 Embedded assembler

The ARM compiler enables you to include assembly code out-of-line, in one or more C or C++ function definitions. Embedded assembler provides unrestricted, low-level access to the target processor, enables you to use the C and C++ preprocessor directives, and gives easy access to structure member offsets.

See the *Assembler Guide* for more information on writing assembly language for the ARM processors.

7.2.1 Embedded assembler syntax

An embedded assembly function definition is marked by the **__asm** (C and C++) or **asm** (C++) function qualifiers, and can be used on:

- member functions
- non-member functions
- template functions
- template class member functions.

Functions declared with **__asm** or **asm** can have arguments, and return a type. They are called from C and C++ in the same way as normal C and C++ functions. The syntax of an embedded assembly function is:

```
__asm return-type function-name(parameter-list)
{
   // ARM/Thumb/Thumb-2 assembler code
   instruction{;comment is optional}
   ...
   instruction
}
```

The initial state of the embedded assembler (ARM or Thumb) is determined by the initial state of the compiler, as specified on the command line. This means that:

- if the compiler starts in ARM state, the embedded assembler uses `--arm`
- if the compiler starts in Thumb state, the embedded assembler uses `--thumb`.

The embedded assembler state at the start of each function is as set by the invocation of the compiler, as modified by `#pragma arm` and `#pragma thumb` pragmas.

You can change the state of the embedded assembler within a function by using explicit `ARM`, `THUMB`, or `CODE16` directives in the embedded assembler function. Such a directive within an `__asm` function does not affect the ARM or Thumb state of subsequent `__asm` functions.

If you are compiling for a Thumb-2 capable processor, you can use Thumb-2 instructions when in Thumb state.

Note

Argument names are permitted in the parameter list, but they cannot be used in the body of the embedded assembly function. For example, the following function uses integer `i` in the body of the function, but this is not valid in assembly:

```
__asm int f(int i)
{
    ADD i, i, #1 // error
}
```

You can use, for example, `r0` instead of `i`.

See the chapter on mixing C, C++, and assembly language in *Developer Guide* for more information on embedded assembly language in C and C++ sources.

Embedded assembler example

Example 7-1 shows a string copy routine as an embedded assembler routine.

Example 7-1 String copy with embedded assembler

```
#include <stdio.h>
__asm void my_strcpy(const char *src, char *dst)
{
loop
      LDRB  r2, [r0], #1
      STRB  r2, [r1], #1
      CMP   r2, #0
      BNE   loop
      BX    lr
}
int main(void)
{
    const char *a = "Hello world!";
    char b[20];
    my_strcpy (a, b);
    printf("Original string: '%s'\n", a);
    printf("Copied   string: '%s'\n", b);
    return 0;
}
```

7.2.2 Restrictions on embedded assembly

The following restrictions apply to embedded assembly functions:

- After preprocessing, `__asm` functions can only contain assembly code, with the exception of the following identifiers (see *Keywords for related base classes* on page 7-22 and *Keywords for member function classes* on page 7-23):

  ```
  __cpp(expr)
  __offsetof_base(D, B)
  __mcall_is_virtual(D, f)
  __mcall_is_in_vbase(D, f)
  __mcall_offsetof_base(D, f)
  __mcall_this_offset(D, f)
  __vcall_offsetof_vfunc(D, f)
  ```

- No return instructions are generated by the compiler for an `__asm` function. If you want to return from an `__asm` function, then you must include the return instructions, in assembly code, in the body of the function.

  Note

  This makes it possible to fall through to the next function, because the embedded assembler guarantees to emit the `__asm` functions in the order you have defined them. However, inlined and template functions behave differently (see *Generation of embedded assembly functions* on page 7-20).

- `__asm` functions do not change the AAPCS rules that apply. This means that all calls between an `__asm` function and a normal C or C++ function must adhere to the AAPCS, even though there are no restrictions on the assembly code that an `__asm` function can use (for example, change state).

7.2.3 Differences between expressions in embedded assembly and C or C++

Be aware of the following differences between embedded assembly and C or C++:

- Assembler expressions are always unsigned. The same expression might have different values between assembler and C or C++. For example:

  ```
  MOV r0, #(-33554432 / 2)        // result is 0x7f000000
  MOV r0, #__cpp(-33554432 / 2)   // result is 0xff000000
  ```

- Assembler numbers with leading zeros are still decimal. For example:

  ```
  MOV r0, #0700                  // decimal 700
  MOV r0, #__cpp(0700)           // octal 0700 == decimal 448
  ```

- Assembler operator precedence differs from C and C++. For example:

```
MOV r0, #(0x23 :AND: 0xf + 1)     // ((0x23 & 0xf) + 1) => 4
MOV r0, #__cpp(0x23 & 0xf + 1)     // (0x23 & (0xf + 1)) => 0
```

- Assembler strings are not null-terminated:

```
DCB "Hello world!"                // 12 bytes (no trailing null)
DCB __cpp("Hello world!")         // 13 bytes (trailing null)
```

Note

The assembler rules apply outside `__cpp`, and the C or C++ rules apply inside `__cpp`. See *The __cpp keyword* on page 7-21.

7.2.4 Generation of embedded assembly functions

The bodies of all the `__asm` functions in a translation unit are assembled as if they are concatenated into a single file that is then passed to the ARM assembler. The order of `__asm` functions in the assembly file that is passed to the assembler is guaranteed to be the same order as in the source file, except for functions that are generated using a template instantiation.

Note

This means that it is possible for control to pass from one `__asm` function to another by falling off the end of the first function into the next `__asm` function in the file, if the return instruction is omitted.

When you invoke `armcc`, the object file produced by the assembler is combined with the object file of the compiler by a partial link that produces a single object file.

The compiler generates an `AREA` directive for each `__asm` function, as in Example 7-2:

Example 7-2 `__asm` function

```
#include <cstddef>
struct X
{
    int x,y;
    void addto_y(int);
};
__asm void X::addto_y(int)
{
    LDR      r2, [r0, #__cpp(offsetof(X, y))]
    ADD      r1, r2, r1
```

```
    STR      r1, [r0, #__cpp(offsetof(X, y))]
    BX       lr
}
```

For this function, the compiler generates:

```
    AREA ||.emb_text||, CODE, READONLY
    EXPORT |_ZN1X7addto_yEi|
#line num "file"
|_ZN1X7addto_yEi| PROC
    LDR r2, [r0, #4]
    ADD r1, r2, r1
    STR r1, [r0, #4]
    BX lr
    ENDP
    END
```

The use of `offsetof` must be inside the `__cpp()` because it is the normal `offsetof` macro from the `cstddef` header file.

Ordinary `__asm` functions are put in an ELF section with the name `.emb_text`. That is, embedded assembly functions are never inlined. However, implicitly instantiated template functions and out-of-line copies of inline functions are placed in an area with a name that is derived from the name of the function, and an extra attribute that marks them as common. This ensures that the special semantics of these kinds of functions is maintained.

Note

Because of the special naming of the area for out-of-line copies of inline functions and template functions, these functions are not in the order of definition, but in an arbitrary order. Therefore, you cannot assume that a code execution falls out of an inline or template function and into another `__asm` function.

7.2.5 The __cpp keyword

You can use the **`__cpp`** keyword to access C or C++ compile-time constant expressions, including the addresses of data or functions with external linkage, from the assembly code. The expression inside the `__cpp` must be a constant expression suitable for use as a C++ static initialization. See *3.6.2 Initialization of non-local objects* and *5.19 Constant expressions* in ISO/IEC 14882:2003.

Example 7-3 on page 7-22 shows a constant replacing the use of `__cpp(expr)`:

Example 7-3 __cpp(*expr*)

```
LDR r0, =__cpp(&some_variable)
LDR r1, =__cpp(some_function)
BL  __cpp(some_function)
MOV r0, #__cpp(some_constant_expr)
```

Names in the __cpp expression are looked up in the C++ context of the __asm function. Any names in the result of a __cpp expression are mangled as required and automatically have IMPORT statements generated for them.

7.2.6 Manual overload resolution

Example 7-4 shows the use of C++ casts to do overload resolution for non-virtual function calls:

Example 7-4 C++ casts

```
void g(int);
void g(long);
struct T
{
    int mf(int);
    int mf(int,int);
};
__asm void  f(T*, int, int)
{
    BL __cpp(static_cast<int (T::*)(int, int)>(&T::mf)) // calls T::mf(int, int)
    BL __cpp(static_cast<void (*)(int)>(g)) // calls g(int)
    BX lr
}
```

7.2.7 Keywords for related base classes

The following keyword enables you to determine the offset from the beginning of an object to a base class sub-object within it:

`__offsetof_base(D, B)`

B must be an unambiguous, non-virtual base class of D.

Returns the offset from the beginning of a D object to the start of the B base subobject within it. The result might be zero. Example 7-5 shows the offset (in bytes) that must be added to a D* p to implement the equivalent of static_cast<B*>(p).

Example 7-5 static_cast<B*>(p)

```
__asm B* my_static_base_cast(D* /*p*/)
{
    if __offsetof_base(D, B) <> 0 // optimize zero offset case
        ADD r0, r0, #__offsetof_base(D, B)
    endif
    BX lr
}
```

These keywords are converted into integer or logical constants in the assembler source. You can only use them in __asm functions, but not in a __cpp expression.

7.2.8 Keywords for member function classes

The following keywords facilitate the calling of virtual and non-virtual member functions from an __asm function. The keywords beginning with __mcall can be used for both virtual and non-virtual functions. The keywords beginning with __vcall can be used only with virtual functions. The keywords do not particularly help in calling static member functions.

For examples of how to use these keywords, see *Calling non-static member functions* on page 7-25.

__mcall_is_virtual(D, f)

Results in {TRUE} if f is a virtual member function found in D, or a base class of D, otherwise {FALSE}. If it returns {TRUE} the call can be done using virtual dispatch, otherwise the call must be done directly.

__mcall_is_in_vbase(D, f)

Results in {TRUE} if f is a non-static member function found in a virtual base class of D, otherwise {FALSE}. If it returns {TRUE} the this adjustment must be done using __mcall_offsetof_vbase(D, f), otherwise it must be done with __mcall_this_offset(D, f).

`__mcall_offsetof_vbase(D, f)`

Where `D` class type and `f` is a non-static member function defined in a virtual base class of `D`, in other words `__mcall_is_in_vbase(D,f)` returns true.

This returns at which negative offset in the vtable of the vtable slot that holds the base offset (from the beginning of a `D` object to the start of the base in which `f` is defined).

This is the `this` adjustment necessary when making a call to `f` with a pointer to a `D`.

Note

The offset returns a positive number that then has to be subtracted from the vtable pointer.

`__mcall_this_offset(D, f)`

Where `D` class type and `f` is a non-static member function defined in `D` or a non-virtual base class of `D`.

This returns the offset from the beginning of a `D` object to the start of the base in which `f` is defined. This is the `this` adjustment necessary when making a call to `f` with a pointer to a `D`. It is either zero if `f` is found in `D` or the same as `__offsetof_base(D,B)`, where `B` is a non-virtual base class of `D` that contains `f`.

If `__mcall_this_offset(D,f)` is used when `f` is found in a virtual base class of `D` it returns an arbitrary value designed to cause an assembly error if used. This is so that such invalid uses of `__mcall_this_offset` can occur in sections of assembly code that are to be skipped.

`__vcall_offsetof_vfunc(D, f)`

Where `D` is a class and `f` is a virtual function defined in `D`, or a base class of `D`.

The function returns the negative offset of the slot in the vtable that holds the base offset. The base offset is calculated as the distance between a `D` object to the start of the base in which `f` is defined.

If `__vcall_offsetof_vfunc(D, f)` is used when `f` is not a virtual member function it returns an arbitrary value designed to cause an assembly error if used.

7.2.9 Calling non-static member functions

You can use keywords beginning with `__mcall` and `__vcall` to call nonvirtual and virtual functions from `__asm` functions. See *Keywords for member function classes* on page 7-23. There is no `__mcall_is_static` to detect static member functions because static member functions have different parameters (that is, no `this`) and, therefore, call sites are likely to already be specific to calling a static member function.

Calling a nonvirtual member function

Example 7-6 shows the following code can be used to call a non-virtual function in either a virtual or non-virtual base:

Example 7-6 Calling a nonvirtual function

```
// rp contains a D* and we want to do the equivalent of rp->f() where f is a
// nonvirtual function
// all arguments other than the this pointer are already setup
// assumes f does not return a struct
 if __mcall_is_in_vbase(D, f)
   LDR r12, [rp]                                  // fetch vtable pointer
   LDR r0, [r12, #-__mcall_offsetof_vbase(D, f)]  // fetch the vbase offset
   ADD r0, r0, rp                                 // do this adjustment
 else
   ADD r0, rp, #__mcall_this_offset(D, f)         // set up and adjust this
                                                  // pointer for D*
 endif
   BL __cpp(&D::f)                                // call D::f
```

Calling a virtual member function

Example 7-7 shows code that can be used to call a virtual function in either a virtual or non-virtual base:

Example 7-7 Calling a virtual function

```
// rp contains a D* and we want to do the equivalent of rp->f() where f is a
// virtual function
// all arguments other than the this pointer are already setup
// assumes f does not return a struct
 if __mcall_is_in_vbase(D, f)
    LDR r12, [rp]                                 // fetch vtable pointer
    LDR r0, [r12, #-__mcall_offsetof_vbase(D, f)] // fetch the base offset
    ADD r0, r0, rp                                // do this adjustment
```

```
   LDR r12, [r0]                                 // fetch vbase vtable pointer
else
   MOV r0, rp                                    // set up this pointer for D*
   LDR r12, [rp]                                 // fetch vtable pointer
   ADD r0, r0, #__mcall_this_offset(D, f)        // do this adjustment
endif
   MOV lr, pc                                    // prepare lr
   LDR pc, [r12, #__vcall_offsetof_vfunc(D, f)]  // calls rp->f()
```

7.3 Legacy inline assembler that accesses sp, lr, or pc

The compilers in *ARM Developer Suite* (ADS) v1.2 and earlier enabled accesses to `sp` (`r13`), `lr` (`r14`), and `pc` (`r15`) from inline assembly code (see *Inline assembler* on page 7-2). Example 7-8 shows how legacy inline assembly code might use `lr`.

Example 7-8 Legacy inline assembly code using lr

```
void func()
{
    int var;
    __asm
    {
        mov  var, lr  /* get the return address of func() */
    }
}
```

There is no guarantee that `lr` contains the return address of a function if your legacy code uses it in inline assembly. For example, there are certain build options or optimizations that might use `lr` for another purpose. The compiler in RVCT v2.0 and later reports an error similar to the following if `lr`, `sp` or `pc` is used in this way:

If you have to access these registers from within a C or C++ source file, you can:

- use embedded assembly (see *Embedded assembler* on page 7-17).
- use the following intrinsics in inline assembly:

  `__current_pc()` To access the `pc` register.

  `__current_sp()` To access the `sp` register.

  `__return_address()` To access the `lr` register.

See also:

- *Accessing sp (r13), lr (r14), and pc (r15) in legacy code*
- *Instruction intrinsics* on page 4-75 in the *Compiler Reference Guide*.

7.3.1 Accessing sp (r13), lr (r14), and pc (r15) in legacy code

The following methods enable you to access the `sp`, `lr`, and `pc` registers correctly in your source code:

Method 1 Use the compiler intrinsics in inline assembly, for example:

```
void printReg()
{
    unsigned int spReg, lrReg, pcReg;
    __asm
    {
        MOV spReg, __current_sp()
        MOV pcReg, __current_pc()
        MOV lrReg, __return_address()
    }
    printf("SP = 0x%X\n",spReg);
    printf("PC = 0x%X\n",pcReg);
    printf("LR = 0x%X\n",lrReg);
}
```

Method 2 Use embedded assembly to access physical ARM registers from within a C or C++ source file, for example:

```
__asm void func()
{
    MOV r0, lr
    ...
    BX lr
}
```

This enables the return address of a function to be captured and displayed, for example, for debugging purposes, to show the call tree.

See *Embedded assembler* on page 7-17.

Note

The compiler might also inline a function into its caller function. If a function is inlined, then the return address is the return address of the function that calls the inlined function. Also, a function might be tail called.

See *__return_address* on page 4-95 in the *Compiler Reference Guide*.

7.4 Differences between inline and embedded assembly code

There are differences between the way inline and embedded assembly is compiled:

- Inline assembly code uses a high-level of processor abstraction, and is integrated with the C and C++ code during code generation. Therefore, the compiler optimizes the C and C++ code, and the assembly code together.
- Unlike inline assembly code, embedded assembly code is assembled separately from the C and C++ code to produce a compiled object that is then combined with the object from the compilation of the C or C++ source.
- Inline assembly code can be inlined by the compiler, but embedded assembly code cannot be inlined, either implicitly or explicitly.

Table 7-1 summarizes the main differences between inline assembler and embedded assembler.

Table 7-1 Differences between inline and embedded assembler

Feature	Embedded assembler	Inline assembler
Instruction set	ARM and Thumb.	ARM only.
ARM assembler directives	All supported.	None supported.
ARMv6 instructions	All supported.	Supports most instructions, with some exceptions, for example `SETEND` and some of the system extensions. The complete set of ARMv6 SIMD instructions is supported.
ARMv7 instructions	All supported.	Not supported.
C/C++ expressions	Constant expressions only.	Full C/C++ expressions.
Optimization of assembly code	No optimization.	Full optimization.
Inlining	Never.	Possible.
Register access	Specified physical registers are used. You can also use `PC`, `LR` and `SP`.	Uses virtual registers (see *Virtual registers* on page 7-8). Using `sp` (`r13`), `lr` (`r14`), and `pc` (`r15`) gives an error. See *Legacy inline assembler that accesses sp, lr, or pc* on page 7-27.
Return instructions	You must add them in your code.	Generated automatically. (The `BX`, `BXJ`, and `BLX` instructions are not supported.)
`BKPT` instruction	Supported directly.	Not supported.

See also *Differences between expressions in embedded assembly and C or C++* on page 7-19.