ARM® Compiler toolchain

Version 5.02

Using the Compiler

ARM®

ARM DUI 0472H (ID070912)

ARM Compiler toolchain
Using the Compiler

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
28 May 2010	A	Non-Confidential	ARM Compiler toolchain v4.1 Release
30 September 2010	B	Non-Confidential	Update 1 for ARM Compiler toolchain v4.1
28 January 2011	C	Non-Confidential	Update 2 for ARM Compiler toolchain v4.1 Patch 3
30 April 2011	D	Non-Confidential	ARM Compiler toolchain v5.0 Release
29 July 2011	E	Non-Confidential	Update 1 for ARM Compiler toolchain v5.0
30 September 2011	F	Non-Confidential	ARM Compiler toolchain v5.01 Release
29 February 2012	G	Non-Confidential	Document update 1 for ARM Compiler toolchain v5.01 Release
27 July 2012	H	Non-Confidential	ARM Compiler toolchain v5.02 Release

Proprietary Notice

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

Contents
ARM Compiler toolchain Using the Compiler

Chapter 1
Conventions and Feedback

The following describes the typographical conventions and how to give feedback:

Typographical conventions

The following typographical conventions are used:

`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.

Feedback on this product

If you have any comments and suggestions about this product, contact your supplier and give:

- your name and company

- the serial number of the product
- details of the release you are using
- details of the platform you are using, 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 documentation

If you have comments on the documentation, e-mail `errata@arm.com`. Give:

- the title
- the number, ARM DUI 0472H
- if viewing online, the topic names to which your comments apply
- if viewing a PDF version of a document, the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

ARM periodically provides updates and corrections to its documentation on the ARM Information Center, together with knowledge articles and *Frequently Asked Questions* (FAQs).

Other information

- ARM Information Center, `http://infocenter.arm.com/help/index.jsp`
- ARM Technical Support Knowledge Articles, `http://infocenter.arm.com/help/topic/com.arm.doc.faqs/index.html`
- ARM Support and Maintenance, `http://www.arm.com/support/services/support-maintenance.php`
- ARM Glossary, `http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html`.

Chapter 2
Overview of the Compiler

The following topics give an overview of the ARM compiler, `armcc`:

2.1 The compiler

The 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. Publications on the C and C++ standards are available from national standards bodies. For example, AFNOR in France and ANSI in the USA.

The compiler also provides a vectorization mode for ARM processors that have NEON™ technology, enabling use of the ARM Advanced *Single Instruction Multiple Data* (SIMD) extension. Vectorization involves the compiler generating NEON vector instructions directly from C or C++ code.

armcc complies with the *Base Standard Application Binary Interface for the ARM Architecture* (BSABI) and generates output objects in ELF with support for *Debug With Arbitrary Record Format* (DWARF) 3 debug tables. armcc uses the *Edison Design Group* (EDG) front end.

Many features of the compiler are designed to take advantage of the target processor or architecture that your code is designed to run on, so knowledge of your target processor or architecture is useful, and in some cases, essential, when working with the compiler.

2.1.1 See also

Concepts

- *NEON technology* on page 4-3.

Reference

- *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)
- the ARM datasheet or technical reference manual for your hardware device.

Other information

- *Application Binary Interface* (ABI) for the ARM Architecture, http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi/index.html
- *DWARF Debugging Standard*, http://www.dwarfstd.org
- *ISO/IEC 9899:1999, C Standard*
- *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.

2.2 Source language modes of the compiler

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

ISO C90 The 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 behavior.

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

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

ISO C++ The 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 in which 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`.

2.2.1 See also

Concepts

- *New language features of C99* on page 6-85
- *Hexadecimal floating-point numbers in C99* on page 6-91.

Reference

Compiler Reference:

- *--c90* on page 3-33
- *--c99* on page 3-34
- *--cpp* on page 3-47
- *--gnu* on page 3-107
- *--strict, --no_strict* on page 3-194
- *Source language modes of the compiler* on page 2-3
- *Language extensions and language compliance* on page 2-7.

2.3 The C and C++ libraries

The following runtime C and C++ libraries are provided:

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 Application Binary Interface for the ARM Architecture* (CLIBABI)
- the *C++ Application Binary Interface for the ARM Architecture* (CPPABI).

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 of the *Standard Template Library* (STL).

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.

2.3.1 See also

Concepts

Using ARM C and C++ Libraries and Floating-Point Support:

- *Compliance with the Application Binary Interface (ABI) for the ARM architecture* on page 2-9
- Chapter 2 *The ARM C and C++ libraries*.

Developing Software for ARM Processors:

- Chapter 5 *Interworking ARM and Thumb*.

Reference

Compiler Reference:

- *The C and C++ libraries* on page 2-10.

Other information

- *Application Binary Interface* (ABI) for the ARM Architecture, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi/index.html`
- *Rogue Wave libraries information*, `http://www.roguewave.com`.

Chapter 3
Getting Started with the Compiler

The following topics outline the command-line options accepted by the ARM compiler, `armcc`. They describe how to invoke the compiler, how to pass options to other tools provided with the compiler, and how to control diagnostic messages:

3.1 Compiler command-line syntax

The command for invoking the compiler is:

`armcc [options] [source]`

where:

options are compiler command-line options that affect the behavior of the compiler.

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.

When you invoke the compiler, you normally specify one or more source files. However, a minority of compiler command-line options do not require you to specify a source file. For example, `armcc --version_number`.

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

`armcc -c [options] input_file_1 ... input_file_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 `--`.

The `-c` option instructs the compiler to perform the compilation step, but not the link step.

3.1.1 See also

Reference

- *Compiler command-line options listed by group* on page 3-4

Compiler Reference:

- *-c* on page 3-31.

Introducing the ARM Compiler toolchain:

- *Compilation tools command-line option rules* on page 2-21.

3.2 Compiler command-line options listed by group

Note

The following characters are interchangeable:

- Nonprefix hyphens and underscores. For example, `--version_number` and `--version-number`.
- Equals signs and spaces. For example, `armcc --cpu=list` and `armcc --cpu list`.

This applies to all tools provided with the compiler.

See the following command-line options in the *Compiler Reference*:

Help

- *--echo* on page 3-83
- *--help* on page 3-112
- *--show_cmdline* on page 3-189
- *--version_number* on page 3-214
- *--vsn* on page 3-219.

Source languages

- *--c90* on page 3-33
- *--c99* on page 3-34
- *--compile_all_input, --no_compile_all_input* on page 3-37
- *--cpp* on page 3-47
- *--gnu* on page 3-107
- *--strict, --no_strict* on page 3-194
- *--strict_warnings* on page 3-195.

Search paths

- *-Idir[,dir,...]* on page 3-114
- *-Jdir[,dir,...]* on page 3-125
- *--kandr_include* on page 3-126
- *--preinclude=filename* on page 3-172
- *--reduce_paths, --no_reduce_paths* on page 3-178
- *--sys_include* on page 3-196.

Project templates

- *--project=filename, --no_project* on page 3-175
- *--reinitialize_workdir* on page 3-179
- *--workdir=directory* on page 3-227.

Precompiled headers

- *--create_pch=filename* on page 3-53
- *--pch* on page 3-165
- *--pch_dir=dir* on page 3-166
- *--pch_messages, --no_pch_messages* on page 3-167
- *--pch_verbose, --no_pch_verbose* on page 3-168
- *--use_pch=filename* on page 3-211.

Preprocessor

- *-C* on page 3-32
- *--code_gen, --no_code_gen* on page 3-35
- *-Dname[(parm-list)][=def]* on page 3-54
- *-E* on page 3-82
- *-M* on page 3-144
- *-Uname* on page 3-206.

C++

- *--anachronisms, --no_anachronisms* on page 3-10
- *--dep_name, --no_dep_name* on page 3-60
- *--export_all_vtbl, --no_export_all_vtbl* on page 3-90
- *--force_new_nothrow, --no_force_new_nothrow* on page 3-94
- *--friend_injection, --no_friend_injection* on page 3-104
- *--guiding_decls, --no_guiding_decls* on page 3-111
- *--implicit_include, --no_implicit_include* on page 3-116
- *--implicit_include_searches, --no_implicit_include_searches* on page 3-117
- *--implicit_typename, --no_implicit_typename* on page 3-119
- *--nonstd_qualifier_deduction, --no_nonstd_qualifier_deduction* on page 3-153
- *--old_specializations, --no_old_specializations* on page 3-158
- *--parse_templates, --no_parse_templates* on page 3-164
- *--pending_instantiations=n* on page 3-169
- *--rtti, --no_rtti* on page 3-185
- *--using_std, --no_using_std* on page 3-212
- *--vfe, --no_vfe* on page 3-215.

Output format

- *--asm* on page 3-24
- *-c* on page 3-31
- *--default_extension=ext* on page 3-59
- *--depend=filename* on page 3-61
- *--depend_format=string* on page 3-63
- *--depend_system_headers, --no_depend_system_headers* on page 3-66
- *--info=totals* on page 3-121
- *--interleave* on page 3-124
- *--list* on page 3-133
- *--md* on page 3-145
- *-o filename* on page 3-154
- *-S* on page 3-187
- *--split_sections* on page 3-193.

Target architectures and processors

- *--arm* on page 3-15
- *--compatible=name* on page 3-36
- *--cpu=list* on page 3-48
- *--cpu=name* on page 3-49
- *--fpu=list* on page 3-99
- *--fpu=name* on page 3-100
- *--thumb* on page 3-197.

Floating-point support

- *--fp16_format=format* on page 3-96
- *--fpmode=model* on page 3-97
- *--fpu=list* on page 3-99
- *--fpu=name* on page 3-100.

Debug

- *--debug, --no_debug* on page 3-56
- *--debug_macros, --no_debug_macros* on page 3-57
- *--dwarf2* on page 3-80
- *--dwarf3* on page 3-81
- *-g* on page 3-105
- *--remove_unneeded_entities, --no_remove_unneeded_entities* on page 3-182.

Code generation

- *--allow_fpreg_for_nonfpdata, --no_allow_fpreg_for_nonfpdata* on page 3-7
- *--alternative_tokens, --no_alternative_tokens* on page 3-9
- *--bigend* on page 3-27
- *--bss_threshold=num* on page 3-30
- *--conditionalize, --no_conditionalize* on page 3-38
- *--dllexport_all, --no_dllexport_all* on page 3-77
- *--dllimport_runtime, --no_dllimport_runtime* on page 3-78
- *--dollar, --no_dollar* on page 3-79
- *--enum_is_int* on page 3-85
- *--exceptions, --no_exceptions* on page 3-87
- *--exceptions_unwind, --no_exceptions_unwind* on page 3-88
- *--export_all_vtbl, --no_export_all_vtbl* on page 3-90
- *--export_defs_implicitly, --no_export_defs_implicitly* on page 3-91
- *--extended_initializers, --no_extended_initializers* on page 3-92
- *--hide_all, --no_hide_all* on page 3-113
- *--littleend* on page 3-137
- *--locale=lang_country* on page 3-138
- *--loose_implicit_cast* on page 3-140
- *--message_locale=lang_country[.codepage]* on page 3-146
- *--min_array_alignment=opt* on page 3-147
- *--multibyte_chars, --no_multibyte_chars* on page 3-149
- *--narrow_volatile_bitfields* on page 3-152
- *--pointer_alignment=num* on page 3-171
- *--protect_stack, --no_protect_stack* on page 3-176
- *--restrict, --no_restrict* on page 3-183
- *--signed_bitfields, --unsigned_bitfields* on page 3-190
- *--signed_chars, --unsigned_chars* on page 3-191
- *--split_ldm* on page 3-192
- *--unaligned_access, --no_unaligned_access* on page 3-207
- *--vectorize, --no_vectorize* on page 3-213
- *--vla, --no_vla* on page 3-218
- *--wchar16* on page 3-224
- *--wchar32* on page 3-225.

Optimization

- *--autoinline, --no_autoinline* on page 3-26
- *--data_reorder, --no_data_reorder* on page 3-55
- *--forceinline* on page 3-95
- *--fpmode=model* on page 3-97
- *--inline, --no_inline* on page 3-122
- *--library_interface=lib* on page 3-128
- *--library_type=lib* on page 3-130
- *--lower_ropi, --no_lower_ropi* on page 3-141
- *--lower_rwpi, --no_lower_rwpi* on page 3-142
- *--ltcg* on page 3-143
- *--multifile, --no_multifile* on page 3-150
- *-Onum* on page 3-156
- *-Ospace* on page 3-160
- *-Otime* on page 3-161
- *--retain=option* on page 3-184.

Note

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

Diagnostics

- *--brief_diagnostics, --no_brief_diagnostics* on page 3-29
- *--diag_error=tag[,tag,...]* on page 3-70
- *--diag_remark=tag[,tag,...]* on page 3-71
- *--diag_style={arm|ide|gnu}* on page 3-72
- *--diag_suppress=tag[,tag,...]* on page 3-73
- *--diag_suppress=optimizations* on page 3-74
- *--diag_warning=tag[,tag,...]* on page 3-75
- *--diag_warning=optimizations* on page 3-76
- *--errors=filename* on page 3-86
- *--remarks* on page 3-181
- *-W* on page 3-220
- *--wrap_diagnostics, --no_wrap_diagnostics* on page 3-228.

Command-line options in a text file

- *--via=filename* on page 3-216.

Linker feedback

- *--feedback=filename* on page 3-93.

Procedure call standard

- *--apcs=qualifer...qualifier* on page 3-11.

Passing options to other tools

- *-Aopt* on page 3-6
- *-Lopt* on page 3-127.

ARM Linux

- *--arm_linux* on page 3-16
- *--arm_linux_configure* on page 3-19
- *--arm_linux_config_file=path* on page 3-18

- *--arm_linux_paths* on page 3-21
- *--configure_gcc=path* on page 3-43
- *--configure_gld=path* on page 3-45
- *--configure_sysroot=path* on page 3-46
- *--configure_cpp_headers=path* on page 3-39
- *--configure_extra_includes=paths* on page 3-40
- *--configure_extra_libraries=paths* on page 3-41
- *--shared* on page 3-188
- *--translate_g++* on page 3-198
- *--translate_gcc* on page 3-200
- *--translate_gld* on page 3-202
- *-Warmcc,option[,option,...]* on page 3-221
- *-Warmcc,--gcc_fallback* on page 3-222.

3.3 Default compiler behavior

The default compiler configuration is determined by the filename extension, for example, *filename*.c, *filename*.cpp, but the command-line options can override this.

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.

armcc can in turn invoke armasm and armlink. For example, if your source code contains embedded assembler, armasm is called. With regard to where armcc looks for the armasm and armlink binaries, it first checks the same location as armcc. If not found, it then checks the PATH locations for these binaries.

3.3.1 See also

Tasks

- *Using stdin to input source code to the compiler* on page 3-12.

Introducing the ARM Compiler toolchain:

- *Using a text file to specify command-line options* on page 2-24.

Concepts

- *Order of compiler command-line options* on page 3-11
- *Filename suffixes recognized by the compiler* on page 3-15
- *Compiler output files* on page 3-17
- *Factors influencing how the compiler searches for header files* on page 3-18
- *Compiler search rules and the current place* on page 3-20
- *The ARMCCnINC environment variable* on page 3-21
- *Compiler command-line options and search paths* on page 3-19
- *PreCompiled Header (PCH) files* on page 5-33.

Introducing the ARM Compiler toolchain:

- *Autocompletion of compilation tools command-line option* on page 2-23
- *TMP and TMPDIR environment variables for temporary file directories* on page 2-27.

Reference

- *Compiler command-line syntax* on page 3-3
- *Compiler command-line options listed by group* on page 3-4.

Introducing the ARM Compiler toolchain:

- *Compilation tools command-line option rules* on page 2-21
- *Specifying command-line options with an environment variable* on page 2-28.

3.4 Order of compiler command-line options

In general, compiler 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 *PreCompiled Header* (PCH) options.

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 makefile or a via file.

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

```
armcc -O1 -O3 -Ospace -Otime ...
```

is executed by the compiler as:

```
armcc -O3 -Otime
```

The environment variable `ARMCC`*n*`_CCOPT` can be used to specify compiler command-line options. Options specified on the command line take precedence over options specified in the environment variable.

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.

3.4.1 See also

Concepts

- *PreCompiled Header (PCH) files* on page 5-33.

Introducing the ARM Compiler toolchain:

- *Specifying command-line options with an environment variable* on page 2-28.

Reference

- *Compiler command-line options listed by group* on page 3-4.

Introducing the ARM Compiler toolchain:

- *Toolchain environment variables* on page 2-14.

3.5 Using `stdin` to input source code to the compiler

Instead of creating a file for your source code, you can use `stdin` to input source code directly on the command line. This is useful if you want to test a short piece of code without having to create a file for it.

To use `stdin` to input source code directly on the command line:

1. Invoke the compiler with the command-line options you want to use. The default compiler mode is C. Use the minus character (`-`) as the source filename to instruct the compiler to take input from `stdin`. For example:

   ```
   armcc --bigend -c -
   ```

   If you want an object file to be written, use the `-o` option. If you want preprocessor output to be sent to the output stream, use the `-E` option. If you want the output to be sent to `stdout`, use the `-o-` option. If you want an assembly listing of the keyboard input to be sent to the output stream after input has been terminated, use none of these options.

2. You cannot input on the same line after the minus character. You must press the return key if you have not already done so.

   The command prompt waits for you to enter more input.

3. Enter your input. For example:

   ```
   #include <stdio.h>
   int main(void)
   { printf("Hello world\n"); }
   ```

4. Terminate your input by entering:
   - `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`.

Otherwise, an object file is created or preprocessor output is sent to the standard output stream, depending on whether you used the `-o` option or the `-E` option.

The compiler accepts source code from the standard input stream in combination with other files, when performing a link step. For example, the following are permitted:

- `armcc -o output.axf - object.o mylibrary.a`
- `armcc -o output.axf --c90 source.c -`

Executing the following command compiles the source code you provide on standard input, and links it into `test.axf`:

```
armcc -o test.axf -
```

You can only combine standard input with other source files when you are linking code. If you attempt to combine standard input with other source files when not linking, the compiler generates an error.

3.5.1 See also

Tasks

Introducing the ARM Compiler toolchain:

- *Using a text file to specify command-line options* on page 2-24.

Reference

- *Compiler command-line syntax* on page 3-3
- *Compiler command-line options listed by group* on page 3-4.

Introducing the ARM Compiler toolchain:

- *Compilation tools command-line option rules* on page 2-21.

3.6 Directing output to `stdout`

If you want output to be sent to the standard output stream, use the `-o-` option. For example:

```
armcc -c -o- hello.c
```

This outputs an assembly listing of the source code to `stdout`.

To send preprocessor output to `stdout`, use the `-E` option.

3.6.1 See also

Tasks

Introducing the ARM Compiler toolchain:

- *Using a text file to specify command-line options* on page 2-24.

Reference

- *Compiler command-line syntax* on page 3-3
- *Compiler command-line options listed by group* on page 3-4.

Introducing the ARM Compiler toolchain:

- *Compilation tools command-line option rules* on page 2-21.

3.7 Filename suffixes recognized by the compiler

The 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 3-1.

Note

Explicitly specifying `--c90`, `--c99`, or `--cpp` overrides the effect of filename suffixes.

Table 3-1 Filename suffixes recognized by the compiler

Suffix	Description	Usage notes
`.c`	C source file	Implies `--c90`
`.C`	C or C++ source file	On UNIX platforms, implies `--cpp`. On non-UNIX platforms, 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.

`.d`	Dependency list file	`.d` is the default output filename suffix for files output using the `--md` option.
`.h`	C or C++ header file	-
`.i`	C or C++ source file	A C or C++ file that has already been preprocessed, and is to be compiled without further preprocessing.
`.ii`	C++ source file	A C++ file that has already been preprocessed, and is to be compiled without further preprocessing.
`.lst`	Error and warning list file	`.lst` is the default output filename suffix for files output using the `--list` option.

`.a`
`.lib`
`.o`
`.obj`
`.so`
	ARM, Thumb, or mixed ARM and Thumb object file or library.	-

`.pch`	Precompiled header file	`.pch` is the default output filename suffix for files output using the `--pch` option.
`.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`.

Table 3-1 Filename suffixes recognized by the compiler (continued)

Suffix	Description	Usage notes
`.S`	ARM, Thumb, or mixed ARM and Thumb assembly language source file.	On UNIX platforms, for files in the input file list suffixed with `.S`, the compiler preprocesses the assembly source prior to passing that source to the assembler. On non-UNIX platforms, `.S` is equivalent to `.s`. That is, preprocessing is not performed.
`.sx`	ARM, Thumb, or mixed ARM and Thumb assembly language source file.	For files in the input file list suffixed with `.sx`, the compiler preprocesses the assembly source prior to passing that source to the assembler.
`.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.

3.7.1 See also

Reference

Compiler Reference:

- *--arm* on page 3-15
- *--c90* on page 3-33
- *--cpp* on page 3-47
- *--interleave* on page 3-124
- *--list* on page 3-133
- *--md* on page 3-145
- *--pch* on page 3-165
- *-S* on page 3-187
- *Implicit inclusion* on page 6-19.

3.8 Compiler output files

By default, output files created by the compiler are located in the current directory. Object files are written in ARM ELF.

3.8.1 See also

Other information

- *ARM ELF File Format*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044-/index.html

3.9 Factors influencing how the compiler searches for header files

Several factors influence how the compiler searches for `#include` header files and source files:

- the value of the environment variable `ARMCC`*`n`*`INC`
- the value of the environment variable `ARMINC`
- 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.

3.9.1 See also

Concepts

- *Compiler search rules and the current place* on page 3-20
- *The ARMCCnINC environment variable* on page 3-21.

Reference

- *Compiler command-line options and search paths* on page 3-19.

Compiler Reference:

- *-Idir[,dir,...]* on page 3-114
- *-Jdir[,dir,...]* on page 3-125
- *--kandr_include* on page 3-126
- *--sys_include* on page 3-196.

Introducing the ARM Compiler toolchain:

- *Toolchain environment variables* on page 2-14.

3.10 Compiler command-line options and search paths

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

Table 3-2 Include file search paths

Compiler option	<include> search order	"include" search order
Neither `-I`*dir*`[,`*dir*`,...]` nor `-J`*dir*`[,`*dir*`,...]`	`ARMCC`*n*`INC`, then `ARMINC`, then `../include`	*Current Working Directory* (CWD) then `ARMCC`*n*`INC`, then `ARMINC`, then `../include`
`-I`*dir*`[,`*dir*`,...]`	`ARMCC`*n*`INC`, then `ARMINC`, then `../include`, then the directory or directories specified by `-I`*dir*`[,`*dir*`,...]`	CWD then the directory or directories specified by `-I`*dir*`[,`*dir*`,...]` then `ARMCC`*n*`INC`, then `ARMINC`, then `../include`
`-J`*dir*`[,`*dir*`,...]`	The directory or directories specified by `-J`*dir*`[,`*dir*`,...]`	CWD then the directory or directories specified by `-J`*dir*`[,`*dir*`,...]`
Both `-I`*dir*`[,`*dir*`,...]` and `-J`*dir*`[,`*dir*`,...]`	The directory or directories specified by `-J`*dir*`[,`*dir*`,...]` and then the directory or directories specified by `-I`*dir*`[,`*dir*`,...]`	CWD then the directory or directories specified by `-I`*dir*`[,`*dir*`,...]` and then the directory or directories specified by `-J`*dir*`[,`*dir*`,...]`
`--sys_include`	No effect	Removes CWD from the search path
`--kandr_include`	No effect	Uses Kernighan and Ritchie search rules

3.10.1 See also

Reference

- *Compiler search rules and the current place* on page 3-20
- *The ARMCCnINC environment variable* on page 3-21.

Compiler Reference:

- *-Idir[,dir,...]* on page 3-114
- *-Jdir[,dir,...]* on page 3-125
- *--kandr_include* on page 3-126
- *--sys_include* on page 3-196.

Introducing the ARM Compiler toolchain:

- *Toolchain environment variables* on page 2-14.

Other information

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

3.11 Compiler search rules and the current place

By default, the compiler uses Berkeley UNIX search rules, so source files and `#include` header files are searched for relative to the *current place*. The current place 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 Kernighan and Ritchie search rules whereby each nonrooted user `#include` is searched for relative to the directory containing the source file that is being compiled.

3.11.1 See also

Concepts

- *Factors influencing how the compiler searches for header files* on page 3-18.

Reference

- *--kandr_include* on page 3-126.

Other information

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

3.12 The ARMCC*n*INC environment variable

The ARMCC*n*INC environment variable points to the location of the included header and source files that are provided with the compilation tools.

This variable might be initialized with the correct path to the header files when the ARM compilation tools are installed or when configured with server modules. You can change this variable, but you must ensure that any changes you make do not break the installation.

The list of directories specified by the ARMCC*n*INC environment variable is colon separated on UNIX and semi-colon separated on Windows, following the convention for the platform you are running on.

If you want to include files from other locations, use the `-I` and `-J` command-line options as required.

When compiling, directories specified with ARMCC*n*INC are searched immediately after directories specified by the `-I` option have been searched, for user include files.

If you use the `-J` option, ARMCC*n*INC is ignored.

3.12.1 See also

Reference

- *Compiler command-line options and search paths* on page 3-19.

Compiler Reference:

- *-Idir[,dir,...]* on page 3-114
- *-Jdir[,dir,...]* on page 3-125.

Introducing the ARM Compiler toolchain:

- *Toolchain environment variables* on page 2-14.

3.13 Code compatibility between separately compiled and assembled modules

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

Interworking qualifiers associated with the `--apcs` compiler command-line option control interworking. Position independence qualifiers, also associated with the `--apcs` compiler command-line option, control position independence, and affect the creation of reentrant and thread-safe code.

Note

This does not mean that you must use the same `--apcs` command-line options to get your modules to work together. You must be familiar with the AAPCS.

3.13.1 See also

Tasks

- *ARM C libraries and multithreading* on page 2-16.

Concepts

Developing Software for ARM Processors:

- Chapter 5 *Interworking ARM and Thumb*.

Reference

Compiler Reference:

- *--apcs=qualifer...qualifier* on page 3-11.

Linker Reference:

- Chapter 10 *BPABI and SysV shared libraries and executables*.

Other information

- *Procedure Call Standard for the ARM Architecture*, `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042-/index.html`

3.14 Linker feedback during compilation

Feedback from the linker to the compiler enables:

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

3.14.1 See also

Tasks

- *Minimizing code size by eliminating unused functions during compilation* on page 3-27
- *Minimizing code size by reducing compilation required for interworking* on page 3-28.

Concepts

- *Unused function code* on page 3-26.

Developing Software for ARM Processors:

- Chapter 5 *Interworking ARM and Thumb*.

3.15 Using GCC fallback when building applications

The ARM Compiler toolchain enables you to build applications that have been developed to build only with GCC. However, there might be cases when the ARM Compiler toolchain cannot complete the build successfully, because of GCC specific functionality that the ARM Compiler toolchain does not emulate. For such cases, a compiler feature is provided that invokes the GCC toolchain to attempt to complete the build. This feature is known as GCC fallback, and it supports the CodeSourcery 2010Q1 gcc toolchain release.

GCC fallback cannot be used to build the Linux kernel, only user applications.

To specify GCC fallback, include the compiler option `-Warmcc,--gcc_fallback`. GCC is invoked with the same GCC-style command-line options that are given to `armcc`. Therefore, GCC fallback has the same effect as the following shell script:

```
armcc $myflags
if found-gcc-specific-coding; then
  gcc $myflags
endif
```

The whole build is still driven by the build script, makefile, or other infrastructure you are using, and that does not change. For example, a single compile step might fail when `armcc` tries to compile that step. `armcc` then attempts to perform that single compile step with gcc. If a link step fails, `armcc` attempts to perform the link with the GCC toolchain, using GNU ld. When `armcc` performs a compile or link step, the include paths, library paths, and Linux libraries it uses are identified in the ARM Linux configuration file. For fallback, you must either:

- use the `--arm_linux_config_file` compiler option to produce the configuration file by configuring `armcc` against an existing `gcc`
- provide an explicit path to `gcc` if you are specifying other configuration options manually.

The GCC toolchain used for fallback is the one that the configuration was created against. Therefore, the paths and libraries used by `armcc` and `gcc` must be equivalent.

If GCC fallback is invoked by `armcc`, a warning message is displayed. If `gcc` also fails, an additional error is displayed, otherwise you get a message indicating that `gcc` succeeded. You also see the original error messages from `armcc` to inform you of the source file or files that failed to compile, and the cause of the problem.

Note

- There is no change to what the ARM Compiler tools link with when using GCC fallback. That is, the tools only link with whatever gcc links with, as identified in the configuration file generated with the `--arm_linux_config_file` compiler option. Therefore, it is your responsibility to ensure that licenses are adhered to, and in particular to check what you are linking with. You might have to explicitly override this if necessary. To do this, include the GNU options `-nostdinc`, `-nodefaultlibs`, and `-nostdlib` on the `armcc` command-line.
- `armcc` invokes the GNU tools in a separate process.
- `armcc` does not optimize any code in any GCC intermediate representations.

To see the commands that are invoked during GCC fallback, specify the `-Warmcc,--echo` command-line option.

The following figure shows a high-level view of the GCC fallback process:

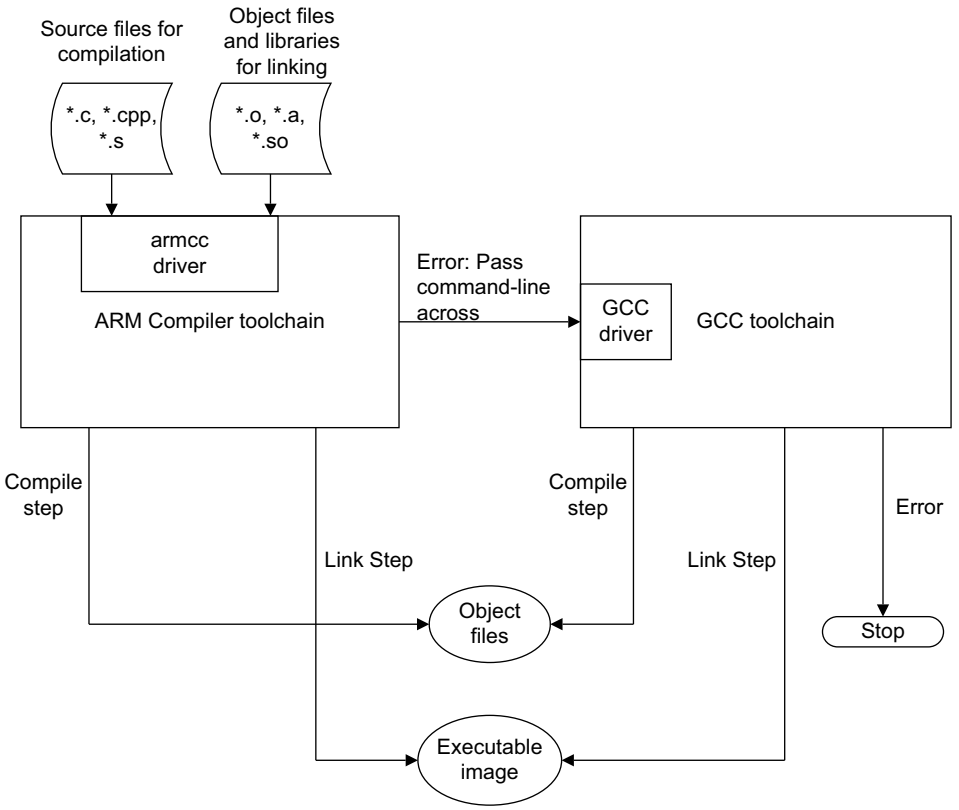

Figure 3-1 GCC fallback process diagram

3.15.1 See also

Reference

Compiler Reference:

- *--arm_linux_config_file=path* on page 3-18
- *--arm_linux_configure* on page 3-19
- *--echo* on page 3-83
- *-Warmcc,option[,option,...]* on page 3-221
- *-Warmcc,--gcc_fallback* on page 3-222.

Other information

- GCC, the GNU Compiler Collection, http://gcc.gnu.org/

3.16 Unused function code

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.
- Where a function is inlined. Where 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.

3.16.1 See also

Tasks

- *Minimizing code size by eliminating unused functions during compilation* on page 3-27.

Concepts

- *Linker feedback during compilation* on page 3-23.

3.17 Minimizing code size by eliminating unused functions during compilation

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. Re-compile using 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.

3.17.1 See also

Tasks

Using the Linker:

- *About linker feedback* on page 5-7.

Concepts

- *Linker feedback during compilation* on page 3-23
- *Unused function code* on page 3-26.

Reference

Compiler Reference:

- *--feedback=filename* on page 3-93.

Linker Reference:

- *--feedback_type=type* on page 2-68.

3.18 Minimizing code size by reducing compilation required for interworking

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.

Note

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

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. Re-compile using 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.

3.18.1 See also

Concepts

- *Linker feedback during compilation* on page 3-23.

Developing Software for ARM Processors:

- Chapter 5 *Interworking ARM and Thumb*.

Reference

Compiler Reference:

- *--feedback=filename* on page 3-93.

Linker Reference:

- *--feedback_type=type* on page 2-68.

Tasks

Using the Linker:

- *About linker feedback* on page 5-7.

3.19 Compilation build time

Modern software applications can comprise many thousands of source code files. These files can take a considerable amount of time to compile. The many different techniques that the ARM compilation tools use to optimize for small code size and high performance can also increase build time.

When you invoke the compiler, the following steps occur:

1. The compiler loads and begins to execute.
2. The compiler tries to obtain a license.
3. The compiler compiles your code.

Loading and beginning to execute the compiler normally takes a fixed period of time.

The time taken to obtain a license does not generally vary if a license is available. However, if a floating license is being used, the time taken to obtain a license depends upon network traffic and whether or not a license is free on the server. In most cases, rather than terminate with error if a license is not immediately available, the compiler waits for a license to become available.

The process of obtaining a floating license is more involved than obtaining a node-locked license. With a node-locked license, the compiler only has to parse the file to check that there is a valid license. With a floating license, the compiler has to check where the license is, send a message through the TCP/IP stacks over the network to the server, then wait for a response. When the compiler receives the response, it then has to check whether or not it has been granted a license. When the compilation is complete, the license has to be returned back to the server.

Floating licenses provide flexibility, but at the cost of speed. If speed is your priority, consider obtaining node-locked licenses for your build machines, or some node-locked licenses locked to USB network cards that can be moved between machines as required.

Setting the environment variable `TCP_NODELAY` to `1` improves FLEX*net* license server system performance when processing license requests. However, you must use this with caution, because it might cause an increase in network traffic.

The time taken to compile your code depends on the size and complexity of the file being compiled. Compiling a small number of large files might be quicker than compiling a larger number of small files. This is because the longer compilation time per file might be offset by the smaller amount of time spent loading and unloading the compiler and obtaining licenses.

3.19.1 See also

Tasks

- *How to minimize compilation build time* on page 3-31
- *Minimizing compilation build time with a single armcc invocation* on page 3-33
- *Minimizing compilation build time with parallel make* on page 3-35.

Concepts

- *Effect of --multifile on compilation build time* on page 3-34
- *Compilation build time and operating system choice* on page 3-36.

Introducing the ARM Compiler toolchain:

- *Licensed features of the toolchain* on page 2-10.

Other information

- *Optimising license checkouts from a floating license server*, http://infocenter.arm.com/help/topic/com.arm.doc.faqs/ka3658.html

3.20 How to minimize compilation build time

To minimize how long the compiler takes to compile your source code, you can:

- Avoid compiling at `-O3` level. `-O3` gives maximum optimization in the code that is generated, but can result in longer build times to achieve such results.
- Precompile header files to avoid repeated compilation.
- Minimize the amount of debug information the compiler generates.
- Guard against multiple inclusion of header files.
- Use the **`restrict`** keyword if you can safely do so, to avoid the compiler having to do compile-time checks for pointer aliasing.
- Try to keep the number of include paths to a minimum. If you have many include paths, ensure that the files you need to include most often exist in directories near the start of the include search path.
- Try compiling a small number of large files instead of a large number of small files. The longer compilation time per file might be offset by less time spent unloading and unloading the compiler and obtaining licenses, particularly if using floating licenses.
- Try compiling multiple files within a single invocation of `armcc` (and single license checkout), instead of multiple `armcc` invocations.
- Floating licenses provide flexibility, but at the cost of speed. Consider obtaining node-locked licenses for your build machines, or some node-locked licenses locked to USB network cards that can be moved between machines as required.
- Consider using or avoiding `--multifile` compilation, depending on the resulting build time.

  Note

  — In RVCT 4.0, if you compile with `-O3`, `--multifile` is enabled by default.

  — In ARM Compiler 4.1 and later, the default is `--multifile` regardless of the optimization level.

- If you are using a makefile-based build environment, consider using a make tool that can apply some form of parallelism.
- Consider your choice of operating system for cross-compilation. Linux generally gives better build speed than Windows, but there are general performance-tuning techniques you can apply on Windows that might help improve build times.

3.20.1 See also

Tasks

- *Methods of reducing debug information in objects and libraries* on page 6-21
- *Minimizing compilation build time with a single armcc invocation* on page 3-33
- *Minimizing compilation build time with parallel make* on page 3-35.

Concepts

- *Compilation build time* on page 3-29
- *PreCompiled Header (PCH) files* on page 5-33
- *Guarding against multiple inclusion of header files* on page 6-22
- *Vectorization on loops containing pointers* on page 4-19

- *Effect of --multifile on compilation build time* on page 3-34
- *Compilation build time and operating system choice* on page 3-36.

Introducing the ARM Compiler toolchain:

- *Licensed features of the toolchain* on page 2-10.

Reference

Compiler Reference:

- *--create_pch=filename* on page 3-53
- *--multifile, --no_multifile* on page 3-150
- *-Onum* on page 3-156
- *--pch* on page 3-165
- *--pch_dir=dir* on page 3-166
- *restrict* on page 4-15.

3.21 Minimizing compilation build time with a single `armcc` invocation

The following type of script incurs multiple loads and unloads of the compiler and multiple license checkouts:

```
armcc file1.c ...
armcc file2.c ...
armcc file3.c ...
```

Instead, you can try modifying your script to compile multiple files within a single invocation of `armcc`. For example, `armcc file1.c file2.c file3.c ...`

For convenience, you can also list all your `.c` files in a single via file invoked with `armcc -via sources.txt`.

Although this mechanism can dramatically reduce license checkouts and loading and unloading of the compiler to give significant improvements in build time, the following limitations apply:

- All files are compiled with the same options.
- Converting existing build systems could be difficult.
- Usability depends on source file structure and dependencies.
- An IDE might be unable to report which file had compilation errors.
- After detecting an error, the compiler does not compile subsequent files.
- Compiling these files at optimization level `-O3` enables multifile compilation, which might increase build times. Use `--no_multifile` to disable multifile compilation.

3.21.1 See also

Tasks

- *How to minimize compilation build time* on page 3-31.

Concepts

- *Compilation build time* on page 3-29.

Introducing the ARM Compiler toolchain:

- *Licensed features of the toolchain* on page 2-10

Reference

Compiler Reference:

- *--multifile, --no_multifile* on page 3-150
- *-Onum* on page 3-156
- *--via=filename* on page 3-216.

3.22 Effect of `--multifile` on compilation build time

When compiling with `--multifile`, the compiler might generate code with additional optimizations by compiling across several source files to produce a single object file. These additional cross-source optimizations can increase compilation time.

Conversely, if there is little additional optimization to apply, and only small amounts of code to check for possible optimizations, then using `--multifile` to generate a single object file instead of several might reduce compilation time as a result of time recovered from creating (opening and closing) multiple object files.

Note

- In RVCT 4.0, if you compile with `-O3`, `--multifile` is enabled by default.
- In ARM Compiler 4.1 and later, `--multifile` is disabled by default, regardless of the optimization level.

3.22.1 See also

Tasks

- *How to minimize compilation build time* on page 3-31.

Concepts

- *Compilation build time* on page 3-29.

Introducing the ARM Compiler toolchain:

- *Licensed features of the toolchain* on page 2-10

Reference

Compiler Reference:

- *--multifile, --no_multifile* on page 3-150
- *-Onum* on page 3-156.

3.23 Minimizing compilation build time with parallel make

If you are using a makefile-based build environment, you could consider using a make tool that can apply some form of parallelism to minimize compilation build time. For example, with GNU `make` you can typically use `make -j N`, where `N` is the number of compile processes you want to have running in parallel. Even on a single machine with a single processor, a performance boost can be achieved. This is because running processes in parallel can hide the effects of network delays and general I/O accesses such as loading and saving files to disk, by fully utilizing the CPU during these times with another compilation process.

If you have multiple processor machines, you can extend the use of parallelism with `make -j N * M`, where `M` is the number of processors.

3.23.1 See also

Tasks

- *How to minimize compilation build time* on page 3-31.

Concepts

- *Compilation build time* on page 3-29.

3.24 Compilation build time and operating system choice

Your choice of operating system can affect compilation build time. Linux generally gives better build speeds than Windows. However, if you are using Windows, there are ways to tune the performance of the operating system at a general level. This might help with increasing the percentage of CPU time that is being used for your build. At a simple level, turning off virus checking software can help, but an Internet search for `"tune windows performance"` provides plenty of information.

3.24.1 See also

Tasks

- *How to minimize compilation build time* on page 3-31.

Concepts

- *Compilation build time* on page 3-29.

Other information

- *On what platforms will my ARM development tools work?*, http://infocenter.arm.com/help/topic/com.arm.doc.faqs/ka3989.html

Chapter 4
Using the NEON Vectorizing Compiler

The following topics provide you with an understanding of the NEON™ unit and explain how to take advantage of automatic vectorizing features:

4.1 NEON technology

ARM NEON technology is a combined 64 and 128-bit advanced *Single Instruction Multiple Data* (SIMD) architecture extension developed by ARM to accelerate the performance of multimedia and signal processing applications. NEON is implemented as part of the ARM 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 32-bit Thumb.

Note

Not all ARM processors support NEON technology. In particular, there is no NEON support for architectures before ARMv7.

4.1.1 See also

Concepts

- *The NEON unit* on page 4-4.

4.2 The NEON unit

The NEON unit has a register bank of thirty-two 64-bit vector registers. The NEON unit can view this register bank as either:

- sixteen 128-bit quadword registers, `Q0` to `Q15`
- thirty-two 64-bit doubleword registers, `D0` to `D31`.

These 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. This is known as vectorization (or more specifically for NEON, *Single Instruction Multiple Data* (SIMD) vectorization).

The NEON unit supports 8-bit, 16-bit and 32-bit integer operations, and some 64-bit operations, in addition to single-precision (32-bit) floating point operations. It can operate on elements in groups of 2, 4, 8, or 16. (The Cortex-A9 processor also supports conversion to and from 16-bit floating-point operations, which the compiler supports when `--fp16_format` is specified, from RVCT 4.0 and later, and ARM Compiler 4.1 and later.)

Note

Vectorization of floating-point code does not always occur automatically. For example, loops that require re-association only vectorize when compiled with `--fpmode fast`. Compiling with `--fpmode fast` enables the compiler to perform some transformations that could affect the result.

The NEON unit is classified as a vector *Single Instruction Multiple Data* (SIMD) unit that operates on multiple elements in a vector register by using one instruction.

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

Table 4-1 Array A

1	2	3	4	5	6	7	8

Array B has the following 8 elements:

Table 4-2 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 4-3 Result

81	72	63	54	45	36	27	18

The NEON unit can only deal with vectors that are stored consecutively in memory, so it is not possible to vectorize indirect addressing.

When writing structures, be aware that NEON structure loads require the structure to contain equal-sized members.

4.2.1 See also

Tasks

- *Generating NEON instructions from C or C++ code* on page 4-7.

Concepts

- *Generating NEON instructions from C or C++ code* on page 4-7
- *Methods of writing code for NEON* on page 4-6.

Reference

Compiler Reference:

- *--fp16_format=format* on page 3-96
- *--fpmode=model* on page 3-97
- *--vectorize, --no_vectorize* on page 3-213.

Other information

- *Introducing NEON Development Article*, `http://infocenter.arm.com/help/topic/com.arm.doc.dht0002-/index.html`

4.3 Methods of writing code for NEON

The following methods of writing code for NEON are available:

- Write in assembly language, or use embedded assembly language in C, and use the NEON instructions directly.
- Write in C or C++ using 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.

Optimizing for performance requires an understanding of where in the program most of the time is spent. To gain maximum performance benefits you might also have to use profiling and benchmarking of the code under realistic conditions.

4.3.1 See also

Tasks

- *Generating NEON instructions from C or C++ code* on page 4-7.

Concepts

- *The NEON unit* on page 4-4
- *NEON C extensions* on page 4-8
- *Automatic vectorization* on page 4-9
- *Data references within a vectorizable loop* on page 4-10
- *Stride patterns and data accesses* on page 4-11
- *Factors affecting NEON vectorization performance* on page 4-12
- *NEON vectorization performance goals* on page 4-13
- *Data dependency conflicts when vectorizing code* on page 4-15
- *Carry-around scalar variables and vectorization* on page 4-17
- *Reduction of a vector to a scalar* on page 4-18
- *Vectorization on loops containing pointers* on page 4-19
- *Nonvectorization on loops containing pointers and indirect addressing* on page 4-21
- *Recommended loop structure for vectorization* on page 4-14
- *Nonvectorization on conditional loop exits* on page 4-22
- *Vectorizable and nonvectorizable use of structures* on page 4-27
- *Vectorization diagnostics to tune code for improved performance* on page 4-32.
- *Vectorizable code example* on page 4-34
- *DSP vectorizable code example* on page 4-37
- *NEON intrinsics provided by the compiler* on page 5-16.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.4 Generating NEON instructions from C or C++ code

You must use RVCT 3.1 or later, or ARM Compiler 4.1 or later, with a valid NEON compiler license.

To generate NEON instructions from C or C++ code, you must specify the following compiler options:

- a target `--cpu` that has NEON capability, for example Cortex-A7, Cortex-A8, Cortex-A9, or Cortex-A15
- `--vectorize` to enable NEON vectorization
- `-O2` (default) or `-O3` optimization level
- `-Otime` to optimize for performance instead of code size.

You can also use `--diag_warning=optimizations` to obtain useful diagnostics from the compiler on what it can and cannot optimize or vectorize. For example:

```
armcc --cpu Cortex-A8 --vectorize -O3 -Otime --diag_warning=optimizations source.c
```

4.4.1 See also

Tasks

- *Selecting the target CPU at compile time* on page 6-8.

Concepts

- *NEON technology* on page 4-3.

Reference

Compiler Reference:

- *--cpu=list* on page 3-48
- *--cpu=name* on page 3-49
- *--vectorize, --no_vectorize* on page 3-213.

Other information

Introducing the ARM Compiler toolchain:

- *Licensed features of the toolchain* on page 2-10.

4.5 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 at C level can be comparable to performance obtained with assembly language coding.

4.5.1 See also

Tasks

- *Generating NEON instructions from C or C++ code* on page 4-7.

Concepts

- *Methods of writing code for NEON* on page 4-6.

Reference

ARM Compiler Reference

- Appendix G *Using NEON Support*.

4.6 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 optimized code gives the same results as nonvectorized code. In certain cases, to avoid the possibility of an incorrect result, vectorization of a loop is not carried out. This can lead to suboptimal code, and you might have to manually tune your code to make it more suitable for automatic vectorization.

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

By coding in vectorizable loops using NEON extensions 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.

4.6.1 See also

Concepts

- *Data references within a vectorizable loop* on page 4-10
- *Factors affecting NEON vectorization performance* on page 4-12
- *Recommended loop structure for vectorization* on page 4-14
- *NEON vectorization performance goals* on page 4-13
- *Data dependency conflicts when vectorizing code* on page 4-15
- *Carry-around scalar variables and vectorization* on page 4-17
- *Reduction of a vector to a scalar* on page 4-18
- *Indicating loop iteration counts to the compiler with __promise(expr)* on page 4-25
- *Vectorization on loops containing pointers* on page 4-19
- *Nonvectorization on loops containing pointers and indirect addressing* on page 4-21
- *Nonvectorization on conditional loop exits* on page 4-22
- *Vectorizable and nonvectorizable use of structures* on page 4-27
- *Vectorization diagnostics to tune code for improved performance* on page 4-32
- *Vectorizable code example* on page 4-34
- *DSP vectorizable code example* on page 4-37
- *Methods of writing code for NEON* on page 4-6.
- *What can limit or prevent automatic vectorization* on page 4-40.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.7 Data references within a vectorizable loop

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

Scalar A single value that does not change throughout all of the loop iterations.

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 4-1 shows the classification of variables in a loop:

`i,j` index variables

`a,b` vectors

`n,x` scalar

Example 4-1 Categorization of a vectorizable loop

```
float *a, *b;
int i, j, n, x;
...
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.

4.7.1 See also

Concepts

- *Stride patterns and data accesses* on page 4-11
- *Automatic vectorization* on page 4-9.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.8 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 pattern, or a stride, of one. A loop that accesses an array with a constant offset between each element used has a constant stride.

Example 4-2 Stride pattern in relation to data access

```
float *a, *b;
int i, j=0, n;
...
for (i = 0; i < n; i++)
{
 /* a is accessed with a stride of 2. */
 /* b is accessed with a stride of 1. */
    *(a+j) = x + b[i];
    j += 2;
};
```

4.8.1 See also

Concepts

- *Data references within a vectorizable loop* on page 4-10.

4.9 Factors affecting NEON vectorization performance

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

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 reduced over more iterations. Tiny iteration counts, such as two or three elements, can be faster to process with nonvector 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.

4.9.1 See also

Concepts

- *NEON vectorization performance goals* on page 4-13
- *Automatic vectorization* on page 4-9.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.10 NEON vectorization 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.

4.10.1 See also

Concepts

- *Factors affecting NEON vectorization performance* on page 4-12
- *Recommended loop structure for vectorization* on page 4-14
- *Data dependency conflicts when vectorizing code* on page 4-15
- *Carry-around scalar variables and vectorization* on page 4-17
- *Reduction of a vector to a scalar* on page 4-18
- *Vectorization on loops containing pointers* on page 4-19
- *Nonvectorization on loops containing pointers and indirect addressing* on page 4-21
- *Nonvectorization on conditional loop exits* on page 4-22
- *Automatic vectorization* on page 4-9.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.11 Recommended loop structure for vectorization

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 that do not contain complex conditional statements or conditional exits. You might have to rewrite your loops to improve the vectorization performance of the code.

4.11.1 See also

Concepts

- *Data dependency conflicts when vectorizing code* on page 4-15
- *Carry-around scalar variables and vectorization* on page 4-17
- *Reduction of a vector to a scalar* on page 4-18
- *Vectorization on loops containing pointers* on page 4-19
- *Nonvectorization on loops containing pointers and indirect addressing* on page 4-21
- *Nonvectorization on conditional loop exits* on page 4-22.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.12 Data dependency conflicts when vectorizing code

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. Detecting data dependencies involving arrays or pointers requires extensive analysis of the arrays used in each loop nest. It also involves 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 Example 4-3, 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 result that differs to the result that is obtained without vectorization, so it is left in its original form.

Example 4-3 Nonvectorizable 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 4-4 vectorizes because the nonvector subscripts of the references to array `a` can never be equal. They 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, so they do not share data.

Example 4-4 Vectorizable data dependency

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

4.12.1 See also

Concepts

- *Recommended loop structure for vectorization* on page 4-14
- *Automatic vectorization* on page 4-9.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.13 Carry-around scalar variables and vectorization

A scalar variable that is used but not set in a loop is replicated in each position in a vector register and the replication is 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 4-5, x is a *used* scalar and y is a *promoted* scalar.

Example 4-5 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 4-6 x is a carry-around scalar.

Example 4-6 Nonvectorizable 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;
};
```

4.13.1 See also

Concepts

- *Recommended loop structure for vectorization* on page 4-14
- *Automatic vectorization* on page 4-9.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.14 Reduction of a vector to a scalar

A special category of scalar use within loops 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:

- the dot product of two vectors
- the maximum value in a vector
- the minimum value in a vector
- the product of all vector elements
- the index of the maximum or minimum element of a vector.

Example 4-7 shows a dot product reduction where `x` is a reduction scalar.

Example 4-7 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 is then reduced into the final resulting scalar.

4.14.1 See also

Concepts

- *Recommended loop structure for vectorization* on page 4-14
- *Automatic vectorization* on page 4-9.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.15 Vectorization on loops containing 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 the **restrict** keyword.

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 4-8.

Example 4-8 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 nonvectorized 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.

In Example 4-8, adding **restrict** to `pa` is sufficient to avoid the runtime test.

4.15.1 See also

Concepts

- *Nonvectorization on loops containing pointers and indirect addressing* on page 4-21
- *Recommended loop structure for vectorization* on page 4-14
- *Automatic vectorization* on page 4-9.

Reference

Compiler Reference:

- *--restrict, --no_restrict* on page 3-183
- *--vectorize, --no_vectorize* on page 3-213
- *restrict* on page 4-15.

4.16 Nonvectorization on loops containing pointers and 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 4-9, `a` is being scattered and `b` is being gathered.

Example 4-9 Nonvectorizable 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.

4.16.1 See also

Concepts

- *Vectorization on loops containing pointers* on page 4-19
- *Recommended loop structure for vectorization* on page 4-14
- *Automatic vectorization* on page 4-9.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.17 Nonvectorization on conditional loop exits

For vectorization purposes, it is best to write loops that do not contain conditional exits from the loop.

Example 4-10 is unable to vectorize because it contains a conditional exit from the loop. In cases like this, you must rewrite the loop, if possible, for vectorization to succeed.

Example 4-10 Nonvectorizable 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;
};
```

4.17.1 See also

Concepts

- *Conditional statements and efficient vectorization* on page 4-31
- *Recommended loop structure for vectorization* on page 4-14
- *Automatic vectorization* on page 4-9.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.18 Vectorizable loop iteration counts

If a loop has a fixed iteration count, automatic vectorization is possible. The iteration count must occur at the start of the loop.

In the vectorizable loop in Table 4-4, the iteration count is `n`. The value of `n` does not change throughout the course of the loop, so this loop can be automatically vectorized.

If a loop does not have a fixed iteration count, automatic vectorization is not possible.

In the nonvectorizable loop in Table 4-4, the value of `i` changes throughout the course of the loop, so this loop cannot be automatically vectorized.

Table 4-4 Vectorizable and nonvectorizable loops

Vectorizable loop

```
/* myprog1.c */
int a[99], b[99], c[99], i, n;
...
for (i = 0; i < n; i++)
{
     a[i] = b[i] + c[i];
}
```

```
armcc --cpu=Cortex-A8 -O3 -Otime
--vectorize myprog1.c -o-
```

```
        ARM
        REQUIRE8
        PRESERVE8

AREA ||.text||, CODE, READONLY, ALIGN=2

f PROC
        PUSH     {r4-r6}
        MOV      r0,#0
        LDR      r4,|L1.160|
        STR      r0,[r4,#0]  ; i
        LDR      r12,[r4,#4]  ; n
        CMP      r12,#0
        BLE      |L1.152|
        ASR      r0,r12,#31
        LDR      r1,|L1.164|
        ADD      r0,r12,r0,LSR #30
        ADD      r2,r1,#0x18c
        ASRS     r0,r0,#2
        SUB      r3,r2,#0x318
        BEQ      |L1.80|

|L1.56|
        VLD1.32  {d0,d1},[r1]!
        SUBS     r0,r0,#1
        VLD1.32  {d2,d3},[r2]!
        VADD.I32 q0,q0,q1
        VST1.32  {d0,d1},[r3]!
        BNE      |L1.56|
```

Nonvectorizable loop

```
/* myprog2.c */
int a[99], b[99], c[99], i, n;
...
while (i < n)
{
     a[i] = b[i] + c[i];
     i += a[i];
};
```

```
armcc --cpu=Cortex-A8 -O3 -Otime
--vectorize myprog2.c
```

```
        ARM
        REQUIRE8
        PRESERVE8

AREA ||.text||, CODE, READONLY, ALIGN=2

||foo|| PROC
        LDR      r3,|L1.76|
        LDR      r0,[r3,#0]  ; i, n
        LDR      r2,[r3,#4]
        CMP      r0,r2
        BXGE     lr
        PUSH     {r4-r6}
        LDR      r12,|L1.80|
        ADD      r4,r12,#0x18c
        SUB      r5,r12,#0x18c

|L1.36|
        LDR      r1,[r12,r0,LSL #2]
        LDR      r6,[r4,r0,LSL #2]
        ADD      r1,r1,r6
        STR      r1,[r5,r0,LSL #2]
        ADD      r0,r0,r1
        CMP      r0,r2
        STR      r0,[r3,#0]  ; i
        BLT      |L1.36|
        POP      {r4-r6}
        BX       lr
        ENDP
```

Table 4-4 Vectorizable and nonvectorizable loops (continued)

Vectorizable loop	Nonvectorizable loop

```
|L1.80|
        AND      r0,r12,#3
        CMP      r0,#0
        BLE      |L1.144|
        SUB      r0,r12,r0
        CMP      r0,r12
        BGE      |L1.144|
        LDR      r1,|L1.164|
        ADD      r2,r1,#0x18c
        SUB      r3,r2,#0x318

|L1.116|
        LDR      r5,[r1,r0,LSL #2]
        LDR      r6,[r2,r0,LSL #2]
        ADD      r5,r5,r6
        STR      r5,[r3,r0,LSL #2]
        ADD      r0,r0,#1
        CMP      r0,r12
        BLT      |L1.116|

|L1.144|
        LDR      r0,[r4,#4]  ; n
        STR      r0,[r4,#0]  ; i

|L1.152|
        POP      {r4-r6}
        BX       lr
        ENDP
```

4.18.1 See also

Concepts

- *Indicating loop iteration counts to the compiler with __promise(expr)* on page 4-25.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.19 Indicating loop iteration counts to the compiler with __promise(*expr*)

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 unknown, 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 code size because additional nonvectorized code is generated to execute any additional loop iterations.

The overhead added by the runtime test is typically insignificant compared with the performance increase that arises from the vectorized code, although corner cases do exist. For example, an iteration count of 17 gives a group of 16 elements to operate on in parallel, with 1 iteration left over as nonvectorized code, whereas an iteration count of 3 gives a group of only 2 elements to operate on in parallel. In the latter case, the overhead of the runtime test is proportionally greater in comparison with the vectorized code.

If you know that the iteration count is divisible by the number of elements that the NEON unit can operate on in parallel, you can indicate this to the compiler using the `__promise` intrinsic, for example:

```
/* Promise the compiler that the loop iteration count is divisible by 16 */
__promise((k % 16) == 0);
for (i = 0; i < k; i++)
{
    ...
}
```

The `__promise` intrinsic is required to enable vectorization if the loop iteration count at the start of the loop is unknown, providing you can make the promise that you claim to make.

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

The disassembled output of Example 4-11 further illustrates the difference that `__promise` makes. The disassembly is reduced to a simple vectorized loop with the removal of nonvectorized code that would otherwise have been required for possible additional loop iterations. That is, loop iterations beyond those that are a multiple of the lanes that can be used for the appropriate data type in a NEON register. (The additional nonvectorized code is known as a scalar fix-up loop. With the use of the `__promise(expr)` intrinsic, the scalar fix-up loop is removed.)

Example 4-11 Using `__promise(expr)` to improve vectorization

```
/* promise.c */

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

When compiling for a processor that supports NEON, the disassembled output might be similar to the following, for example:

```
armcc -S promise.c

        ARM
        REQUIRE8
        PRESERVE8
```

```
        AREA ||.text||, CODE, READONLY, ALIGN=2


f PROC
        VMOV.I32 q0,#0x1
        ASR      r1,r1,#2
|L0.8|
        VLD1.32  {d2,d3},[r0]
        SUBS     r1,r1,#1
        VADD.I32 q1,q1,q0
        VST1.32  {d2,d3},[r0]!
        BNE      |L0.8|
        BX       lr
        ENDP
```

4.19.1 See also

Concepts

- *Automatic vectorization* on page 4-9
- *Vectorizable loop iteration counts* on page 4-23.

Reference

Compiler Reference:

- *__promise intrinsic* on page 5-144
- *--restrict, --no_restrict* on page 3-183
- *restrict* on page 4-15
- *--vectorize, --no_vectorize* on page 3-213.

4.20 Vectorizable and nonvectorizable use of structures

See the following vectorizable and nonvectorizable uses of structures:

- *Grouping use of structures for vectorization* on page 4-28
- *struct member lengths and vectorization* on page 4-29.

4.21 Grouping use of structures for vectorization

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. See Example 4-12 and Example 4-13.

Example 4-12 Structure access resulting in a nonvectorizable loop

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

Example 4-13 Structure access resulting in a vectorizable loop

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

4.21.1 See also

Concepts

- *Vectorizable and nonvectorizable use of structures* on page 4-27.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.22 `struct` member lengths and vectorization

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 4-14.

Example 4-14 Nonvectorizable code caused by inconsistent structure member lengths

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

The code in Example 4-14 could be rewritten for vectorization by using the same data type throughout the structure. For example, if the variable `b` is to be of type **`int`**, consider making variables `a` and `c` of type **`int`** rather than **`short`**.

4.22.1 See also

Concepts

- *Vectorizable and nonvectorizable use of structures* on page 4-27.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.23 Nonvectorization of function calls to non-inline functions from within loops

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

Splitting complex operations into several functions to aid clarity is common practice. However, if such functions are to be considered for vectorization, they must be marked with the `__inline` or `__forceinline` keywords if they are called from within any loops. These functions are then expanded inline for vectorization to take place.

4.23.1 See also

Concepts

- *Recommended loop structure for vectorization* on page 4-14.

Reference

Compiler Reference:

- *__forceinline* on page 5-10
- *__inline* on page 5-13
- *--vectorize, --no_vectorize* on page 3-213.

4.24 Conditional statements and efficient vectorization

For efficient vectorization, loops must contain mostly assignment statements and must 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 4-15 shows an acceptable use of conditional statements.

Example 4-15 Vectorizable condition

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

4.24.1 See also

Concepts

- *Nonvectorization on conditional loop exits* on page 4-22
- *Recommended loop structure for vectorization* on page 4-14.

Reference

Compiler Reference:

- *--vectorize, --no_vectorize* on page 3-213.

4.25 Vectorization diagnostics to tune code for improved performance

The compiler can provide diagnostic information to indicate where vectorization optimizations are successfully applied and where it failed to apply vectorization. The command-line options that provide this information are `--diag_warning=optimizations` and `--remarks`.

Example 4-16 shows two functions that implement a simple sum operation on an array. This code does not vectorize.

Example 4-16 Nonvectorizable 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);
    /* Function calls cannot be vectorized */
}
```

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

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. Similarly, `--remarks`.

The compiler must generate a runtime test for aliasing and output both vectorized and scalar copies of the code. Example 4-17 shows how this can be improved using the **`restrict`** keyword if you know that the pointers are not aliased.

Example 4-17 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 4-17, 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 4-18 on page 4-33 shows the final improvement that can be made to obtain the best performance from vectorization.

Example 4-18 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;
    __promise((n % 4) == 0);
    /* n is a multiple of 4 */
    for(i = 0; i < (n & ~3); i++) *(pa + i) = addition(*(pb + i),x);
}
```

4.25.1 See also

Concepts

- *Vectorizable code example* on page 4-34
- *DSP vectorizable code example* on page 4-37
- *Automatic vectorization* on page 4-9.

Reference

Compiler Reference:

- *--diag_suppress=tag[,tag,...]* on page 3-73
- *--diag_warning=tag[,tag,...]* on page 3-75
- *--restrict, --no_restrict* on page 3-183
- *restrict* on page 4-15
- *--vectorize, --no_vectorize* on page 3-213.

4.26 Vectorizable code example

Example 4-19 shows a complete vectorizable code example. The options required to build this example are listed within the introductory source code comments. The `--cpu=name` option must name a processor that has NEON technology, for example, Cortex-A7, Cortex-A8, Cortex-A9, or Cortex-A15.

`--diag_warning=optimizations` can be used to view where vectorization optimization is applied.

The use of `__promise` enables the compiler to generate smaller and faster code. The code still works and vectorizes without these promises, but is then larger and slower.

Example 4-19 Vectorizable 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
```

4.26.1 See also

Concepts

- *Embedded assembler support in the compiler* on page 8-36
- *DSP vectorizable code example* on page 4-37
- *Vectorization diagnostics to tune code for improved performance* on page 4-32.

Reference

Compiler Reference:

- *--cpu=name* on page 3-49
- *--diag_suppress=tag[,tag,...]* on page 3-73
- *--diag_warning=tag[,tag,...]* on page 3-75
- *Predefined macros* on page 5-183
- *--restrict, --no_restrict* on page 3-183
- *restrict* on page 4-15
- *--vectorize, --no_vectorize* on page 3-213.

Linker Reference:

- *--entry=location* on page 2-58.

4.27 DSP vectorizable code example

Example 4-20 shows a complete *Digital Signal Processing* (DSP) vectorizable code example. The options required to build this example are listed within the introductory source code comments. The `--cpu=name` option must name a processor that has NEON technology, for example, Cortex-A7, Cortex-A8, Cortex-A9, or Cortex-A15.

`--diag_warning=optimizations` can be used to view where vectorization optimization is applied.

Example 4-20 DSP vectorizable 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 -O3 -Otime --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
```

4.27.1 See also

Concepts

- *Embedded assembler support in the compiler* on page 8-36
- *Vectorizable code example* on page 4-34
- *Vectorization diagnostics to tune code for improved performance* on page 4-32.

Reference

Compiler Reference:

- *--cpu=name* on page 3-49
- *--diag_suppress=tag[,tag,...]* on page 3-73
- *--diag_warning=tag[,tag,...]* on page 3-75
- *Predefined macros* on page 5-183
- *--restrict, --no_restrict* on page 3-183
- *restrict* on page 4-15
- *--vectorize, --no_vectorize* on page 3-213.

Linker Reference:

- *--entry=location* on page 2-58.

4.28 What can limit or prevent automatic vectorization

Table 4-5 summarizes what can limit or prevent automatic vectorization of loops.

Table 4-5 Factors that limit or prevent automatic vectorization

Inhibiting factor	Extent to which it applies
Not having a valid NEON compiler license.	You require a valid NEON compiler license to generate NEON instructions.
Source code without loops.	Automatic vectorization involves loop analysis. Without loops, automatic vectorization cannot apply.
Target processor.	The target processor (`--cpu`) must have NEON capability if NEON instructions are to be generated. For example, Cortex-A7, Cortex-A8, Cortex-A9, or Cortex-A15.
Floating-point code.	Vectorization of floating-point code does not always occur automatically. For example, loops that require re-association only vectorize when compiled with `--fpmode fast`.
`--no_vectorize` by default.	By default, generation of NEON vector instructions directly from C or C++ code is disabled, and must be enabled with `--vectorize`.
`-O`*`time`* not specified.	`-O`*`time`* must be specified to reduce execution time and enable loops to vectorize.
`-O`*`num`* not set high enough.	The optimization level you set must be `-O2` or `-O3`. Loops do not vectorize at `-O0` or `-O1`.
Risk of incorrect results.	If there is a *risk* of an incorrect result, vectorization is not applied where that risk occurs. You might have to manually tune your code to make it more suitable for automatic vectorization.
Earlier manual optimization attempts.	Automatic vectorization can be impeded by earlier manual optimization attempts. For example, manual loop unrolling in the source code, or complex array accesses.
No vector access pattern.	If variables in a loop lack a vector access pattern, the compiler cannot automatically vectorize the loop.
Data dependencies between different iterations of a loop.	Where there is a possibility of the use and storage of arrays overlapping on different iterations of a loop, there is a data dependency problem. A loop cannot be safely vectorized if the vector order of operations can change the results, so the compiler leaves the loop in its original form or only partially vectorizes the loop.
Memory hierarchy.	Performing relatively few arithmetic operations on large data sets retrieved from main memory is limited by the memory bandwidth of the system. Most processors are relatively unbalanced between memory bandwidth and processor capacity. This can adversely affect the automatic vectorization process.

Table 4-5 Factors that limit or prevent automatic vectorization (continued)

Inhibiting factor	Extent to which it applies
Iteration count not fixed at start of loop.	For automatic vectorization, it is generally best to write simple loops with iterations that are fixed at the start of the loop. If a loop does not have a fixed iteration count, automatic addressing is not possible.
Conditional loop exits.	It is best to write loops that do not contain conditional exits from the loop.
Carry-around scalar variables.	Carry-around scalar variables are a problem for automatic vectorization because the value computed in one pass of the loop is carried forward into the next pass.
`__promise(expr)` not used.	Failure to use `__promise(expr)` where it could make a difference to automatic vectorization can limit automatic vectorization.
Pointer aliasing.	Pointer aliasing prevents the use of automatically vectorized code.
Indirect addressing.	Indirect addressing is not vectorizable because the NEON unit can only deal with vectors stored consecutively in memory.
Separating access to different parts of a structure into separate loops.	Each part of a structure must be accessed within the same loop for automatic vectorization to occur.
Inconsistent length of members within a structure.	If members of a structure are not all the same length, the compiler does not attempt to use vector loads.
Calls to non-inline functions.	Calls to non-inline functions from within a loop inhibits vectorization. If such functions are to be considered for vectorization, they must be marked with the **`__inline`** or **`__forceinline`** keywords.
`if` and `switch` statements.	Extensive use of `if` and `switch` statements can affect the efficiency of automatic vectorization.

You can use `--diag_warning=optimizations` to obtain compiler diagnostics on what can and cannot be vectorized.

4.28.1 See also

Concepts

- *Automatic vectorization* on page 4-9
- *Data references within a vectorizable loop* on page 4-10
- *Factors affecting NEON vectorization performance* on page 4-12
- *Recommended loop structure for vectorization* on page 4-14
- *Nonvectorization on conditional loop exits* on page 4-22
- *Data dependency conflicts when vectorizing code* on page 4-15
- *Carry-around scalar variables and vectorization* on page 4-17
- *Indicating loop iteration counts to the compiler with __promise(expr)* on page 4-25
- *Vectorization on loops containing pointers* on page 4-19
- *Nonvectorization on loops containing pointers and indirect addressing* on page 4-21
- *Vectorizable loop iteration counts* on page 4-23

- *Grouping use of structures for vectorization* on page 4-28
- *struct member lengths and vectorization* on page 4-29
- *Nonvectorization of function calls to non-inline functions from within loops* on page 4-30
- *Conditional statements and efficient vectorization* on page 4-31
- *Vectorization diagnostics to tune code for improved performance* on page 4-32.

Introducing the ARM Compiler toolchain:

- *Licensed features of the toolchain* on page 2-10.

Reference

Compiler Reference:

- *--cpu=list*
- *--cpu=name* on page 3-49
- *--diag_warning=optimizations* on page 3-76
- *__forceinline* on page 5-10
- *--fpmode=model* on page 3-97
- *__inline* on page 5-13
- *-Onum* on page 3-156
- *-Otime* on page 3-161
- *__promise intrinsic* on page 5-144
- *restrict* on page 4-15
- *--vectorize, --no_vectorize* on page 3-213.

Chapter 5
Compiler Features

The following topics give an overview of ARM-specific features of the compiler:

5.1 Compiler intrinsics

The C and C++ languages are suited to a wide variety of tasks but they do not provide in-built 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. However, often 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 5-1 shows its implementation in C.

Example 5-1 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;
}
```

Compiler intrinsics are functions provided by the compiler. They enable you to easily incorporate domain-specific operations in C and C++ source code without resorting to complex implementations in assembly language. Using compiler intrinsics, you can achieve more complete coverage of target architecture instructions than you would from the instruction selection of the compiler.

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 5-1 has the form:

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

5.1.1 See also

Concepts

- *Performance benefits of compiler intrinsics* on page 5-5
- *ARM assembler instruction intrinsics supported by the compiler* on page 5-6
- *European Telecommunications Standards Institute (ETSI) basic operations* on page 5-12
- *Texas Instruments (TI) C55x intrinsics for optimizing C code* on page 5-15.

Reference

Compiler Reference:

- *Instruction intrinsics* on page 5-117
- *ETSI basic operations* on page 5-169

- *C55x intrinsics* on page 5-171
- Appendix G *Using NEON Support*.

5.2 Performance benefits of compiler intrinsics

The use of compiler 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 instruction 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.

5.2.1 See also

Concepts

- *Compiler intrinsics* on page 5-3.

5.3 ARM assembler instruction intrinsics supported by the compiler

See the following topics for a range of instruction intrinsics for generating 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.

5.4 Generic intrinsics supported by the compiler

In the *Compiler Reference,* see the following generic intrinsics that are ARM language extensions to the ISO C and C++ standards:

- *__breakpoint intrinsic* on page 5-118
- *__current_pc intrinsic* on page 5-122
- *__current_sp intrinsic* on page 5-123
- *__nop intrinsic* on page 5-138
- *__return_address intrinsic* on page 5-150
- *__semihost intrinsic* on page 5-153.

Implementations of these intrinsics are available across all architectures.

5.4.1 See also

Reference

- *GNU built-in functions* on page 5-180.

5.5 Compiler intrinsics for controlling IRQ and FIQ interrupts

The intrinsics `__disable_irq`, `__enable_irq`, `__disable_fiq` and `__enable_fiq` control IRQ and FIQ interrupts.

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, as follows:

- 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, or if `--compatible` is being used, the compiler calls a helper function, for example:

  ```
  BL    __ARM_disable_irq
  ```

5.5.1 See also

Reference

Compiler Reference:

- *__disable_fiq intrinsic* on page 5-124
- *__disable_irq intrinsic* on page 5-125
- *__enable_fiq intrinsic* on page 5-127
- *__enable_irq intrinsic* on page 5-128.

5.6 Compiler intrinsics for inserting optimization barriers

The 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 optimization barrier intrinsics `__schedule_barrier`, `__force_stores` and `__memory_changed` do not generate code, but they can result in slightly increased code size and additional memory accesses.

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. See the Technical Reference Manual for your ARM processor for more information.

5.6.1 See also

Reference

Compiler Reference:

- *__force_stores intrinsic* on page 5-131
- *__memory_changed intrinsic* on page 5-137
- *__schedule_barrier intrinsic* on page 5-152.

Other information

The *Technical Reference Manual* for your ARM processor.

5.7 Compiler intrinsics for inserting native instructions

The following intrinsics insert ARM processor instructions into the instruction stream generated by the compiler:

Reference

Compiler Reference:

- *__cdp intrinsic* on page 5-119
- *__clrex intrinsic* on page 5-120
- *__ldrex intrinsic* on page 5-132
- *__ldrt intrinsic* on page 5-135
- *__pld intrinsic* on page 5-141
- *__pli intrinsic* on page 5-143
- *__rbit intrinsic* on page 5-148
- *__rev intrinsic* on page 5-149
- *__ror intrinsic* on page 5-151
- *__sev intrinsic* on page 5-155
- *__strex intrinsic* on page 5-159
- *__strt intrinsic* on page 5-162
- *__swp intrinsic* on page 5-163
- *__wfe intrinsic* on page 5-165
- *__wfi intrinsic* on page 5-166
- *__yield intrinsic* on page 5-167.

5.8 Compiler intrinsics for *Digital Signal Processing* (DSP)

The compiler provides intrinsics that assist in the implementation of DSP algorithms. These intrinsics introduce the appropriate target instructions for:

- ARM, on architectures from ARMv5TE onwards
- Thumb, on architectures with Thumb-2 technology.

Not every instruction has its own intrinsic. The compiler can combine several intrinsics, or combinations of intrinsics and C operators to generate more powerful instructions. For example, the ARMv5TE `QDADD` instruction is generated by a combination of `__qadd` and `__qdbl`.

5.8.1 See also

Reference

Compiler Reference:

- *__clz intrinsic* on page 5-121
- *__fabs intrinsic* on page 5-129
- *__fabsf intrinsic* on page 5-130
- *__qadd intrinsic* on page 5-145
- *__qdbl intrinsic* on page 5-146
- *__qsub intrinsic* on page 5-147
- *__sqrt intrinsic* on page 5-156
- *__sqrtf intrinsic* on page 5-157
- *__ssat intrinsic* on page 5-158
- *__usat intrinsic* on page 5-164
- *ARMv6 SIMD intrinsics* on page 5-168.

5.9 *European Telecommunications Standards Institute* (ETSI) basic operations

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 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 the ARM compilation tools.

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++.

5.9.1 See also

Concepts

- *Compiler support for European Telecommunications Standards Institute (ETSI) basic operations* on page 5-13.

Reference

Compiler Reference:

- *ETSI basic operations* on page 5-169.

Other information

- 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)*.

5.10 Compiler support for *European Telecommunications Standards Institute* (ETSI) basic operations

ARM Compiler 4.1 and later provide support for the ETSI basic operations through the header file `dspfns.h`. The `dspfns.h` header file contains definitions of the ETSI basic operations as a combination of C code and intrinsics.

See `dspfns.h` for a complete list of the ETSI basic operations supported in ARM Compiler 4.1 and later.

ARM Compiler 4.1 and later support the original ETSI family of basic operations as 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.

5.10.1 See also

Concepts

- *European Telecommunications Standards Institute (ETSI) basic operations* on page 5-12.

Reference

Compiler Reference:

- *ETSI basic operations* on page 5-169.

5.11 Overflow and carry status flags for C and C++ code

The implementation of the *European Telecommunications Standards Institute* (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.

5.12 *Texas Instruments* (TI) C55x intrinsics for optimizing C code

The TI C55x compiler recognizes a number of intrinsics for the optimization of C code. The ARM compilation tools support the emulation of selected TI C55x intrinsics through the header file, `c55x.h`.

`c55x.h` gives a complete list of the TI C55x intrinsics that are emulated by the ARM compilation tools.

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.
- 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`.

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

- 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 the ARM compilation tools 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`.

5.12.1 See also

Other information

- Texas Instruments, http://www.ti.com/.

5.13 NEON intrinsics provided by the compiler

As an alternative to automatic compiler vectorization, NEON intrinsics provide an intermediate step between a vectorizing compiler and writing assembly language for SIMD code generation. This feature makes it easier to write code that takes advantage of the NEON architecture when compared to writing assembly language directly.

NEON intrinsics are defined in the header file `arm_neon.h`. The header file defines both the intrinsics and a set of vector types.

5.13.1 See also

Tasks

- *Using NEON intrinsics* on page 5-17.

Concepts

- *The NEON unit* on page 4-4
- *Methods of writing code for NEON* on page 4-6.

Reference

Compiler Reference:

- Appendix G *Using NEON Support*.

5.14 Using NEON intrinsics

To build an example that uses NEON intrinsics:

1. Create the following example C program source code:

Example 5-2 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;
}
```

2. Compile the example source code with the following options:

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

3. Link the resulting object file using the following command:

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

4. Use a compatible debugger to load and run the resulting image `neon_example.axf`.

5.14.1 See also

Concepts

- *NEON intrinsics provided by the compiler* on page 5-16.

Reference

Compiler Reference:

- *-c* on page 3-31
- *--cpu=name* on page 3-49
- *--debug, --no_debug* on page 3-56
- Appendix G *Using NEON Support*.

5.15 Compiler support for accessing registers using 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 following declaration declares `R0` as a named register variable for the register `r0`:

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

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). Example 3-3 shows the use of named register variables to set the saturation flag `Q` in the APSR.

Example 5-3 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 */
```

```
/* Declare PSR as a register variable for the "apsr" register */
register PSR apsr __asm("apsr");

void set_Q(void)
{
    apsr.b.Q = 1;
}
```

See also Example 5-4.

Example 5-4 Clearing the Q flag in the APSR using a named register variable

```
register unsigned int _apsr __asm("apsr");

__forceinline void ClearQFlag(void)
{
    _apsr = _apsr & ~0x08000000; // clear Q flag
}
```

Disassembly:

```
ClearQFlag
    MRS    r0,APSR ; formerly CPSR
    BIC    r0,r0,#0x80000000
    MSR    APSR_nzcvq,r0; formerly CPSR_f
    BX     lr
```

See also Example 5-5.

Example 5-5 Setting up stack pointers using named register variables

```
register unsigned int _control __asm("control");
register unsigned int _msp     __asm("msp");
register unsigned int _psp     __asm("psp");

void init(void)
{
    _msp = 0x30000000;        // set up Main Stack Pointer
    _control = _control | 3;  // switch to User Mode with Process Stack
    _psp = 0x40000000;        // set up Process Stack Pointer
}
```

Disassembly, compiled using `--cpu=7-M`:

```
init
    MOV    r0,0x30000000
    MSR    MSP,r0
    MRS    r0,CONTROL
    ORR    r0,r0,#3
    MSR    CONTROL,r0
    MOV    r0,#0x40000000
    MSR    PSP,r0
    BX     lr
```

You can also use named register variables to access registers within a coprocessor. The string syntax within the declaration corresponds to how you intend to use the variable. For example, to declare a variable that you intend to use with the MCR instruction, look up the instruction syntax for this instruction and use this syntax when you declare your variable. See Example 5-6.

Example 5-6 Setting bits in a coprocessor register using a named register variable

```
register unsigned int PMCR __asm("cp15:0:c9:c12:0");

__inline void __reset_cycle_counter(void)
{
    PMCR = 4;
}

Disassembly:

__reset_cycle_counter PROC
    MOV    r0,#4
    MCR    p15,#0x0,r0,c9,c12,#0      ; move from r0 to c9
    BX     lr
    ENDP
```

In Example 5-6, PMCR is declared as a register variable of type **unsigned int**, that is associated with the cp15 coprocessor, with CRn = c9, CRm = c12, opcode1 = 0, and opcode2 = 0 in an MCR or MRC instruction. The MCR encoding in the disassembly corresponds with the register variable declaration.

The physical coprocessor register is specified with a combination of the two register numbers, CRn and CRm, and two opcode numbers. This maps to a single physical register.

The same principle applies if you want to manipulate individual bits in a register, but you write normal variable arithmetic in C, and the compiler does a read-modify-write of the coprocessor register. See Example 5-7.

Example 5-7 Manipulating bits in a coprocessor register using a named register variable

```
register unsigned int SCTLR __asm("cp15:0:c1:c0:0");

/* Set bit 11 of the system control register */
void enable_branch_prediction(void)
{
    SCTLR |= (1 << 11);
}

Disassembly:

__enable_branch_prediction PROC
    MRC    p15,#0x0,r0,c1,c0,#0
    ORR    r0,r0,#0x800
    MCR    p15,#0x0,r0,c1,c0,#0
    BX     lr
    ENDP
```

5.15.1 See also

Reference

Assembler Reference:

- *Application Program Status Register* on page 3-18
- *MCR, MCR2, MCRR, and MCRR2* on page 3-94.

Compiler Reference:

- *__asm* on page 5-9
- *Named register variables* on page 5-176.

5.16 Pragmas recognized by the compiler

See the following topics in the *Compiler Reference*:

Pragmas for saving and restoring the pragma state

- *#pragma pop* on page 5-109
- *#pragma push* on page 5-110.

Pragmas controlling optimization goals

- *#pragma Onum* on page 5-103
- *#pragma Ospace* on page 5-105
- *#pragma Otime* on page 5-106.

Pragmas controlling code generation

- *#pragma arm* on page 5-87
- *#pragma thumb* on page 5-112
- *#pragma exceptions_unwind, #pragma no_exceptions_unwind* on page 5-95.

Pragmas controlling loop unrolling:

- *#pragma unroll [(n)]* on page 5-113
- *#pragma unroll_completely* on page 5-115.

Pragmas controlling *PreCompiled Header* (PCH) processing

- *#pragma hdrstop* on page 5-97
- *#pragma no_pch* on page 5-102.

Pragmas controlling anonymous structures and unions

- *#pragma anon_unions, #pragma no_anon_unions* on page 5-86.

Pragmas controlling diagnostic messages

- *#pragma diag_default tag[,tag,...]* on page 5-90
- *#pragma diag_error tag[,tag,...]* on page 5-91
- *#pragma diag_remark tag[,tag,...]* on page 5-92
- *#pragma diag_suppress tag[,tag,...]* on page 5-93
- *#pragma diag_warning tag[, tag, ...]* on page 5-94.

Miscellaneous pragmas

- *#pragma arm section [section_type_list]* on page 5-88
- *#pragma import(__use_full_stdio)* on page 5-99
- *#pragma inline, #pragma no_inline* on page 5-101
- *#pragma once* on page 5-104
- *#pragma pack(n)* on page 5-107
- *#pragma softfp_linkage, #pragma no_softfp_linkage* on page 5-111
- *#pragma import symbol_name* on page 5-98.

5.17 Compiler and processor support for bit-banding

Bit-banding is supported by the compiler in the following ways:

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

Bit-banding is a feature of the Cortex-M3 and Cortex-M4 processors (`--cpu=Cortex-M3` and `--cpu=Cortex-M4`) and some derivatives (for example, `--cpu=Cortex-M3-rev0`). This functionality is not available on other ARM processors.

5.17.1 See also

Concepts

- *Compiler type attribute, __attribute__((bitband))* on page 5-25
- *--bitband compiler command-line option* on page 5-27
- *How the compiler handles bit-band objects placed outside bit-band regions* on page 5-29.

Reference

Compiler Reference:

- *__attribute__((bitband)) type attribute* on page 5-65
- *--bitband* on page 3-28.

Other information

- *Technical Reference Manual* for your processor.

5.18 Compiler type attribute, __attribute__((bitband))

__attribute__((bitband)) is a type attribute that is used to bit-band type definitions of structures.

In Example 5-8, the unplaced bit-banded objects must be relocated into the bit-band region. This can be achieved by either using an appropriate scatter-loading description file or by using the --rw_base linker command-line option.

Example 5-8 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 */
```

Alternatively, __attribute__((at())) can be used to place bit-banded objects at a particular address in the bit-band region. See Example 5-9.

Example 5-9 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 */
```

5.18.1 See also

Concepts

- *Compiler and processor support for bit-banding* on page 5-24.

Using the Linker:

- Chapter 8 *Using scatter files*.

Reference

Compiler Reference:

- *__attribute__((at(address))) variable attribute* on page 5-72
- *__attribute__((bitband)) type attribute* on page 5-65
- *--bitband* on page 3-28.

Linker Reference:

- *--rw_base=address* on page 2-139.

Other information

- *Technical Reference Manual* for your processor.

5.19 --bitband compiler command-line option

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

When --bitband is applied to foo.c in Example 5-10, the write to value.i is bit-banded. That is, the 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 5-10 Using the --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 5-11, the access to rts is bit-banded.

Example 5-11 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 */
```

5.19.1 See also

Concepts

- *Compiler and processor support for bit-banding* on page 5-24.

Reference

Compiler Reference:

- *__attribute__((at(address))) variable attribute* on page 5-72
- *__attribute__((bitband)) type attribute* on page 5-65
- *--bitband* on page 3-28.

Other information

- *Technical Reference Manual* for your processor.

5.20 How the compiler handles bit-band objects placed outside bit-band regions

Bit-band objects must not be placed outside bit-band regions. If you do inadvertently place a bit-band object outside a bit-band region, either using the `at` attribute, or using an integer pointer to a particular address, the compiler responds as follows:

- If the `bitband` attribute is applied to an object type and `--bitband` is not specified on the command line, the compiler generates an error.
- If the `bitband` attribute is applied to an object type and `--bitband` is specified on the command line, the compiler generates a warning, and ignores the request to bit-band.
- If the `bitband` attribute is *not* applied to an object type and `--bitband` is specified on the command line, the compiler ignores the request to bit-band.

5.20.1 See also

Concepts

- *Compiler and processor support for bit-banding* on page 5-24.

Reference

Compiler Reference:

- *__attribute__((at(address))) variable attribute* on page 5-72
- *__attribute__((bitband)) type attribute* on page 5-65
- *--bitband* on page 3-28.

5.21 Compiler support for thread-local storage

Thread-local storage 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.

5.21.1 See also

Reference

Compiler Reference:

- *__declspec(thread)* on page 5-38.

5.22 Compiler eight-byte alignment features

The 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 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 language 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.
- In RVCT v2.0 and later, and in ARM Compiler 4.1 and later, `double` and `long long` data types are eight-byte aligned for compliance with the *Application Binary Interface for the ARM Architecture* (AEABI). 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.

5.22.1 See also

Concepts

Using the Linker:

- *Section alignment with the linker* on page 4-22.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *alloca()* on page 2-5.

Assembler Reference:

- *Alignment restrictions in load and store, element and structure instructions* on page 4-15.

Other information

- *Procedure Call Standard for the ARM Architecture*, `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042-/index.html`
- *Application Binary Interface (ABI) for the ARM Architecture*, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi/index.html`

5.23 Using compiler and linker support for symbol versions

The compiler and the linker both 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. Use this extension to rename the function symbol into a symbol that has either of the following names:

- *function*`@@`*ver* for a default *ver* of *function*
- *function*`@`*ver* for a nondefault *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 nondefault version:

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

5.23.1 See also

Concepts

Using the Linker:

- *Symbol versioning for BPABI models* on page 10-11.

Reference

Compiler Reference:

- *Assembler labels* on page 4-39.

5.24 *PreCompiled Header* (PCH) files

When you compile 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 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. In such a case, 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 mean that the compiler might not always be able to use the PCH file. Address Space Randomization on *Red Hat Enterprise Linux 3* (RHE3) is one example of a possible system configuration issue.

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

5.24.1 See also

Tasks

- *Manually specifying the filename and location of a PreCompiled Header (PCH) file* on page 5-40
- *Selectively applying PreCompiled Header (PCH) file processing* on page 5-41
- *Suppressing PreCompiled Header (PCH) file processing* on page 5-42.

Concepts

- *Automatic PreCompiled Header (PCH) file processing* on page 5-34
- *PreCompiled Header (PCH) file processing and the header stop point* on page 5-35
- *PreCompiled Header (PCH) file creation requirements* on page 5-36
- *Compilation with multiple PreCompiled Header (PCH) files* on page 5-38
- *Obsolete PreCompiled Header (PCH) files* on page 5-39
- *Message output during PreCompiled Header (PCH) processing* on page 5-43
- *Performance issues with PreCompiled Header (PCH) files* on page 5-44.

Reference

Compiler Reference:

- *--pch* on page 3-165
- *--pch_dir=dir* on page 3-166
- *--pch_messages, --no_pch_messages* on page 3-167
- *--pch_verbose, --no_pch_verbose* on page 3-168
- *#pragma hdrstop* on page 5-97
- *#pragma no_pch* on page 5-102.

5.25 Automatic *PreCompiled Header* (PCH) file processing

The `--pch` command-line option enables automatic PCH file processing. 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 the `--pch_dir` option is specified.

5.25.1 See also

Concepts

- *Order of compiler command-line options* on page 3-11
- *PreCompiled Header (PCH) file processing and the header stop point* on page 5-35
- *PreCompiled Header (PCH) file creation requirements* on page 5-36
- *PreCompiled Header (PCH) files* on page 5-33.

Reference

Compiler Reference:

- *--pch* on page 3-165
- *--pch_dir=dir* on page 3-166.

5.26 *PreCompiled Header* (PCH) file processing and 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;
```

The header stop point can be manually specified with `#pragma hdrstop`. If used, this pragma must be placed before the first token that does not belong to a preprocessing directive. In this example, it must be placed before `int`, as follows:

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

#pragma hdrstop
int i;
```

If a `#if` block encloses the first non-preprocessor token or `#pragma hdrstop`, 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 /* Header stop point lies immediately before #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`.

5.26.1 See also

Tasks

- *Selectively applying PreCompiled Header (PCH) file processing* on page 5-41
- *Selectively applying PreCompiled Header (PCH) file processing* on page 5-41
- *Suppressing PreCompiled Header (PCH) file processing* on page 5-42.

Concepts

- *PreCompiled Header (PCH) files* on page 5-33
- *Automatic PreCompiled Header (PCH) file processing* on page 5-34
- *PreCompiled Header (PCH) file creation requirements* on page 5-36
- *PreCompiled Header (PCH) file creation requirements* on page 5-36
- *Performance issues with PreCompiled Header (PCH) files* on page 5-44.

Reference

Compiler Reference:

- *#pragma hdrstop* on page 5-97
- *--pch* on page 3-165
- *--pch_dir=dir* on page 3-166.

5.27 *PreCompiled Header* (PCH) file creation requirements

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

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

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

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

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

- The header stop point must not be inside a `#if` block or a `#define` that is started within a header file.
- The processing that precedes the header stop point must not have produced any errors.

  Note

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

- No references to predefined macros `__DATE__` or `__TIME__` must appear.
- No instances of the `#line` preprocessing directive must appear.
- `#pragma no_pch` must not appear.
- The code preceding the header stop point must have introduced a sufficient number of declarations to justify the overhead associated with precompiled headers.

5.27.1 See also

Tasks

- *Selectively applying PreCompiled Header (PCH) file processing* on page 5-41
- *Suppressing PreCompiled Header (PCH) file processing* on page 5-42.

Concepts

- *PreCompiled Header (PCH) files* on page 5-33
- *Automatic PreCompiled Header (PCH) file processing* on page 5-34
- *PreCompiled Header (PCH) file processing and the header stop point* on page 5-35
- *PreCompiled Header (PCH) file creation requirements*
- *Performance issues with PreCompiled Header (PCH) files* on page 5-44.

Reference

Compiler Reference:

- *#pragma hdrstop* on page 5-97
- *#pragma no_pch* on page 5-102
- *--pch* on page 3-165
- *--pch_dir=dir* on page 3-166.
- *--pch_messages, --no_pch_messages* on page 3-167
- *--pch_verbose, --no_pch_verbose* on page 3-168.

5.28 Compilation with multiple *PreCompiled Header* (PCH) files

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 example, 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 is created if the requirements for PCH file creation are met.

5.28.1 See also

Concepts

- *PreCompiled Header (PCH) files* on page 5-33.

5.29 Obsolete *PreCompiled Header* (PCH) files

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.

5.29.1 See also

Concepts

- *PreCompiled Header (PCH) files* on page 5-33.

5.30 Manually specifying the filename and location of a *PreCompiled Header* (PCH) file

To manually specify the filename and location of a PCH file, use any of the following compiler command-line options:

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

If you use `--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.

Note

If multiple options are specified on the same command line, the following rules apply:

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

5.30.1 See also

Concepts

- *PreCompiled Header (PCH) files* on page 5-33
- *Automatic PreCompiled Header (PCH) file processing* on page 5-34.

Reference

Compiler Reference:

- *--create_pch=filename* on page 3-53
- *--pch* on page 3-165
- *--pch_dir=dir* on page 3-166
- *--use_pch=filename* on page 3-211.

5.31 Selectively applying *PreCompiled Header* (PCH) file processing

You can selectively include and exclude header files for PCH file processing, even if you are using automatic PCH file processing. Do this by inserting a manual header stop point using the `#pragma hdrstop` directive in the primary source file. Insert it 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.

5.31.1 See also

Tasks

- *Suppressing PreCompiled Header (PCH) file processing* on page 5-42.

Concepts

- *PreCompiled Header (PCH) files* on page 5-33
- *Automatic PreCompiled Header (PCH) file processing* on page 5-34
- *Pragmas recognized by the compiler* on page 5-23.

Reference

Compiler Reference:

- *#pragma hdrstop* on page 5-97
- *#pragma no_pch* on page 5-102.

5.32 Suppressing *PreCompiled Header* (PCH) file processing

To suppress PCH file processing, use the `#pragma no_pch` directive in the primary source file. You do not have to place this directive at the beginning of the file for it to take effect. For example, no PCH file is created if you compile the following source code with `armcc --create_pch=foo.pch myprog.c`:

```
#include "xxx.h"
#pragma no_pch
#include "zzz.h"
```

If you want to selectively enable PCH processing, for example, subject `xxx.h` to PCH file processing, but not `zzz.h`, replace `#pragma no_pch` with `#pragma hdrstop`, as follows:

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

5.32.1 See also

Tasks

- *Selectively applying PreCompiled Header (PCH) file processing* on page 5-41.

Concepts

- *PreCompiled Header (PCH) files* on page 5-33
- *Automatic PreCompiled Header (PCH) file processing* on page 5-34
- *Pragmas recognized by the compiler* on page 5-23.

Reference

Compiler Reference:

- *#pragma hdrstop* on page 5-97
- *#pragma no_pch* on page 5-102.

5.33 Message output during *PreCompiled Header* (PCH) processing

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

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

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

A verbose mode is available using `--pch_verbose`. In verbose mode, the compiler displays a message for each PCH file that is considered but cannot be used, giving the reason why it cannot be used.

5.33.1 See also

Concepts

- *PreCompiled Header (PCH) files* on page 5-33.

Reference

Compiler Reference:

- *--pch_messages, --no_pch_messages* on page 3-167
- *--pch_verbose, --no_pch_verbose* on page 3-168.

5.34 Performance issues with *PreCompiled Header* (PCH) files

Typically, the overhead of creating and reading a PCH file is small, even for reasonably large header files. If the PCH 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.

5.34.1 See also

Concepts

- *PreCompiled Header (PCH) files* on page 5-33.

5.35 Default compiler options that are affected by optimization level

In general, optimization levels are independent from the default behavior of command-line options. However, there are a small number of exceptions where the level of optimization you use changes the default option.

These exceptions are:

- `--autoinline`, `--no_autoinline`
- `--data_reorder`, `--no_data_reorder`.

Depending on the value of `-O`*num* you use (`-O0`, `-O1`, `-O2`, or `-O3`), the default option changes as specified. See the individual command-line option reference descriptions for more information.

5.35.1 See also

Reference

Compiler Reference:

- *--autoinline, --no_autoinline* on page 3-26
- *--data_reorder, --no_data_reorder* on page 3-55
- *-Onum* on page 3-156.

Chapter 6
Compiler Coding Practices

The 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, you can increase the portability, efficiency and robustness of your C and C++ source code. The following topics describe some of these programming practices, together with some programming techniques that are specific to ARM processors:

6.1 The compiler as an optimizing compiler

The 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, and the correct target processor and architecture.

Note

Optimization options can limit debug information generated by the compiler.

6.1.1 See also

Concepts

- *Compiler optimization for code size versus speed* on page 6-6
- *Compiler optimization levels and the debug view* on page 6-7
- *Selecting the target CPU at compile time* on page 6-8
- *Optimization of loop termination in C code* on page 6-9
- *Compiler optimization and the volatile keyword* on page 6-13.

Reference

Compiler Reference:

- *--autoinline, --no_autoinline* on page 3-26
- *--cpu=name* on page 3-49
- *--data_reorder, --no_data_reorder* on page 3-55
- *--forceinline* on page 3-95
- *--fpmode=model* on page 3-97
- *--inline, --no_inline* on page 3-122
- *--library_interface=lib* on page 3-128
- *--library_type=lib* on page 3-130
- *--lower_ropi, --no_lower_ropi* on page 3-141
- *--lower_rwpi, --no_lower_rwpi* on page 3-142
- *--ltcg* on page 3-143
- *--multifile, --no_multifile* on page 3-150
- *-Onum* on page 3-156
- *-Ospace* on page 3-160
- *-Otime* on page 3-161
- *--retain=option* on page 3-184.

6.2 Compiler optimization for code 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.

6.2.1 See also

Reference

Compiler Reference:

- *-Ospace* on page 3-160
- *-Otime* on page 3-161.

6.3 Compiler 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 with `-Ospace` and `-Otime` generally impacts the debug view.

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 to 15%. To reduce the size of your debug tables, use the `--remove_unneeded_entities` option.

6.3.1 See also

Tasks

- *Methods of reducing debug information in objects and libraries* on page 6-21.

Concepts

- *Benefits of reducing debug information in objects and libraries* on page 6-20.

Reference

Compiler Reference:

- *--debug, --no_debug* on page 3-56
- *--debug_macros, --no_debug_macros* on page 3-57
- *--dwarf2* on page 3-80
- *--dwarf3* on page 3-81
- *-Onum* on page 3-156
- *-Ospace* on page 3-160
- *-Otime* on page 3-161
- *--remove_unneeded_entities, --no_remove_unneeded_entities* on page 3-182.

Other information

- *ARM ELF File Format*, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0101-/index.html`

6.4 Selecting the target CPU at compile time

Each new version of the ARM architecture typically supports extra instructions, extra modes of operation, pipeline differences, and register renaming. You can often significantly improve the performance of your C or C++ code by selecting the appropriate target processor at compile time.

To specify a target processor at compile time:

1. Decide whether the compiled program is to run on a specific ARM architecture-based processor or on different ARM processors.
2. Obtain the name, or names, of the target processors recognized by the compiler using the following compiler command-line option:

   ```
   --cpu=list
   ```

3. If the compiled program is to run on a specific ARM architecture-based processor, having obtained the name of the processor with the `--cpu=list` option, select the target processor using the `--cpu=`*`name`* compiler command-line option. For example, to compile code to run on a Cortex-A9 processor:

   ```
   armcc --cpu=Cortex-A9 myprog.c
   ```

   Alternatively, if the compiled program is to run on different ARM processors, choose the lowest common denominator architecture appropriate for the application and then specify that architecture in place of the processor name. For example, to compile code for processors supporting the ARMv6 architecture:

   ```
   armcc --cpu=6 myprog.c
   ```

Selecting the target processor using the `--cpu=`*`name`* command-line option 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.

`--cpu=list` lists all the processors and architectures supported by the compiler.

6.4.1 See also

Reference

Compiler Reference:

- *--cpu=list* on page 3-48
- *--cpu=name* on page 3-49.

6.5 Optimization of loop termination in C code

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.

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

- use simple termination conditions
- write count-down-to-zero loops
- use counters of type **`unsigned int`**
- test for equality against zero.

Following any or all of these guidelines, separately or in combination, is likely to result in better code.

Table 6-1 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 6-1 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 6-2 shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations of Table 6-1, where the C code for both implementations has been compiled using the options `-O2 -Otime`.

Table 6-2 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 6-2 shows that the `ADD` and `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 used instead.

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. It is even more important if the original termination condition involves a function call. For example:

```
for (...; i < get_limit(); ...);
```

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.

6.5.1 See also

Concepts

- *Loop unrolling in C code* on page 6-11.

6.6 Loop unrolling in C code

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.

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 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 6-3. 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 implementation unrolled four times, with an optimization applied by combining the four shifts of `n` into one shift.

Unrolling frequently provides new opportunities for optimization.

Table 6-3 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 6-4 shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations of Table 6-3 on page 6-11, where the C code for each implementation has been compiled using the option -O2.

Table 6-4 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 ARM9, 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 per loop iteration, taking on average only three cycles per bit. However, the cost is the larger code size of fifteen instructions.

6.6.1 See also

Concepts

- *Optimization of loop termination in C code* on page 6-9.

6.7 Compiler optimization and the `volatile` keyword

Higher optimization levels can reveal problems in some programs that are not apparent at lower optimization levels, for example, missing **`volatile`** qualifiers. This can manifest itself in a number of ways. Code might become stuck in a loop while polling hardware, multi-threaded code might exhibit strange behavior, or optimization might result in the removal of code that implements deliberate timing delays. In such cases, it is possible that some variables are required to be declared as **`volatile`**.

The declaration of 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, by another thread of execution such as an interrupt routine or signal handler, or by hardware. Because the value of a **`volatile`**-qualified variable can change at any time, the actual 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 its value in a register to avoid memory accesses. Similarly, when used in the context of implementing a sleep or timer delay, declaring a variable as **`volatile`** tells the compiler that a specific type of behavior is intended, and that such code must not be optimized in such a way that it removes the intended functionality.

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

The use of the **`volatile`** keyword is illustrated in the two sample routines of Table 6-5. Both of these routines loop reading a buffer until a status flag `buffer_full` is set to true. The state of `buffer_full` can change asynchronously with program flow.

The two versions of the routine differ only in the way that `buffer_full` is declared. The first routine version is incorrect. Notice that the variable `buffer_full` is not qualified as **`volatile`** in this version. In contrast, the second version of the routine shows the same loop where `buffer_full` is correctly qualified as **`volatile`**.

Table 6-5 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 6-6 shows the corresponding disassembly of the machine code produced by the compiler for each of the sample versions in Table 6-1 on page 6-9, where the C code for each implementation has been compiled using the option `-O2`.

Table 6-6 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 6-6, the statement `LDR r0, [r0, #0]` loads the value of `buffer_full` into register `r0` outside the loop labeled `|L1.12|`. 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 `|L1.12|`.

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 `|L1.8|`. As a result, the loop `|L1.8|` 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 or signal handler.

The compiler does not optimize the variables you have declared as volatile.

6.8 Code metrics

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

- measure code and data sizes
- generate dynamic callgraphs
- measure stack use.

6.8.1 See also

Concepts

- *Stack use in C and C++* on page 6-17.

Reference

- *Code metrics for measurement of code size and data size* on page 6-16.

6.9 Code metrics for measurement of code size and data size

The following command-line options enable you to measure code and data size:

- `--info=sizes` (`armlink` and `fromelf`)
- `--info=totals` (`armcc`, `armlink`, and `fromelf`)
- `--map` (`armlink`).

6.9.1 See also

Reference

Compiler Reference:

- *--info=totals* on page 3-121.

Linker Reference:

- *--info=topic[,topic,...]* on page 2-80
- *--map, --no_map* on page 2-108.

Using the fromelf Image Converter:

- *--info=topic[,topic,...]* on page 4-47.

6.10 Stack use in C and C++

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 *ARM Architecture Procedure Call Standard* (AAPCS), for instance, when register contents are saved on entry into subroutines
- local variables, including local arrays, structures, unions, and in C++, classes.

Some stack usage is not obvious, such as:

- Local integer or floating point variables are allocated stack memory if they are spilled (that is, not allocated to a register).
- Structures are normally allocated to the stack. A space equivalent to `sizeof(struct)` padded to a multiple of four bytes is reserved on the stack. The compiler tries to allocate structures to registers instead.
- Arrays are always allocated to the stack. Again, a space equivalent to `sizeof(struct)` padded to a multiple of four bytes is reserved on the stack.
- Several optimizations can introduce new temporary variables to hold intermediate results. The optimizations include: CSE elimination, live range splitting and structure splitting. The compiler tries to allocate these temporary variables to registers. If not, it spills them to the stack.
- Generally, 16-bit Thumb code makes more use of the stack than ARM code and 32-bit Thumb code, because 16-bit Thumb code has only eight registers available for allocation, compared to fourteen for ARM code and 32-bit Thumb code.
- The AAPCS mandates that some function arguments are passed through the stack instead of the registers, depending on their type, size, and order.

6.10.1 Methods of estimating stack usage

Stack use is difficult to estimate because it is code dependent, and can vary between runs depending on the code path that the program takes on execution. However, it is possible to manually estimate the extent of stack utilization using the following methods:

- 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 the debugger to set a watchpoint on the last available location in the stack and see if the watchpoint is ever hit.

Note

Running your program under a debug monitor like a *Real-Time System Model* (RTSM), in DS-5 Debugger or RealView Debugger, has a severe performance penalty, because the watched address is checked for every instruction. Using DSTREAM or RealView ICE and RealView Trace has no such penalty.

- Use the debugger, and:
  1. Allocate space in memory for the stack that is much larger than you expect to require.
  2. Fill the stack space with copies of a known value, for example, `0xDEADDEAD`.
  3. Run your application, or a fixed portion of it. Aim to use as much of the stack space as possible in the test run. For example, try to execute the most deeply nested function calls and the worst case path found by the static analysis. Try to generate interrupts where appropriate, so that they are included in the stack trace.
  4. After your application has finished executing, examine the stack space of memory to see how many of the known values have been overwritten. The space has garbage in the used part and the known values in the remainder.
  5. Count the number of garbage values and multiply by four, to give their size, in bytes.

  The result of the calculation shows how the size of the stack has grown, in bytes.
- Use RTSM, and define a region of memory where access is not allowed directly below your stack in memory, with a map file. If the stack overflows into the forbidden region, a data abort occurs, which can be trapped by the debugger.

6.10.2 Methods of reducing stack usage

In general, you can lower the stack requirements of your program by:

- writing small functions that only require a small number of variables
- avoiding the use of large local structures or arrays
- avoiding recursion, for example, by using an alternative algorithm
- minimizing the number of variables that are in use at any given time at each point in a function
- using C block scope and declaring variables only where they are needed, so overlapping the memory used by distinct scopes.

The use of C block scope involves declaring variables only where they are required. This minimizes use of the stack by overlapping memory required by distinct scopes.

Note

Code performance is optimized by locating the stack in fast (zero wait-state), on-chip, 32-bit RAM. The ARM (`LDMFD` and `STMFD`) and Thumb (`PUSH` and `POP`) stack access instructions both push and pop a number of 32-bit registers on or off the stack. If the stack is in 32-bit memory, each register access takes one cycle. However, if the stack is in 16-bit memory then each register access takes two cycles, reducing overall performance.

6.10.3 See also

Reference

Linker Reference:

- *--callgraph, --no_callgraph* on page 2-26
- *--info=topic[,topic,...]* on page 2-80.

Using the fromelf Image Converter:

- *--info=topic[,topic,...]* on page 4-47.

Other information

- *ARM DS-5 Using the Debugger*, http://infocenter.arm.com/help/topic/com.arm.doc.dui0446-/index.html
- *RealView Debugger User Guide*, http://infocenter.arm.com/help/topic/com.arm.doc.dui0153-/index.html
- *Getting Started with DS-5, ARM DS-5 Product Overview, About Real-Time System Models*, http://infocenter.arm.com/help/topic/com.arm.doc.dui0478-/CACDACDJ.html
- *RealView Development Suite Real-Time System Models User Guide*, http://infocenter.arm.com/help/topic/com.arm.doc.dui0424-/index.html
- *Real-Time System Model Reference*, http://infocenter.arm.com/help/topic/com.arm.doc.dui0575-/index.html
- *Procedure Call Standard for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042-/index.html.

6.11 Benefits of 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.

6.11.1 See also

Concepts

- *Methods of reducing debug information in objects and libraries* on page 6-21.

6.12 Methods of reducing debug information in objects and libraries

There are a number of 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 required.
- Guard against the multiple inclusion of header files. Place multiple-inclusion guards inside the header file, rather than around the `#include` statement. For example, if you have a header file `foo.h`, 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.
- Consider using (or not using) `--remove_unneeded_entities`.

  Caution

  Although `--remove_unneeded_entities` can help to reduce the amount of debug information generated per file, it has the disadvantage of reducing the number of debug sections that are common to many files. This reduces the number of common debug sections that the linker is able to remove at final link time, and can result in a final debug image that is larger than necessary. For this reason, use `--remove_unneeded_entities` only when necessary.

6.12.1 See also

Tasks

- *Guarding against multiple inclusion of header files* on page 6-22.

Concepts

- *Benefits of reducing debug information in objects and libraries* on page 6-20.

Reference

Compiler Reference:

- *--debug_macros, --no_debug_macros* on page 3-57
- *--remarks* on page 3-181.

6.13 Guarding against multiple inclusion of header files

Guarding against multiple inclusion of header files:

- improves compilation time
- reduces the size of object files generated using the `-g` compiler command-line option, which can speed up link time
- avoids compilation errors that arise from including the same code multiple times.

For example:

```
/* foo.h */
#ifndef FOO_H
#define FOO_H 1
...
#endif

/* bar.c */
#ifndef FOO_H
#include "foo.h"
#endif
```

6.13.1 See also

Reference

Compiler Reference:

- *-g* on page 3-105.

6.14 Methods of minimizing function parameter passing overhead

There are a number of 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 when using software floating-point.
- Avoid functions with a variable number of parameters. Functions taking a variable number of arguments effectively pass all their arguments on the stack.

6.14.1 See also

Reference

Compiler Reference:

- *Basic data types* on page 6-5.

6.15 Functions that return multiple values through registers

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 multiple values to be returned 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;
```

`__value_in_regs` can be used anywhere where multiple values have to be returned 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`.

6.15.1 See also

Concepts

- *Methods of minimizing function parameter passing overhead* on page 6-23.

Reference

Compiler Reference:

- *__value_in_regs* on page 5-26.

6.16 Functions that return the same result when called with the same arguments

A function that always returns the same result when called with the same arguments, and does not change any global data, is referred to as a pure function.

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.

Using the keyword **`__pure`** when declaring a function indicates that the function is a pure function.

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 *Common Subexpression Eliminations* (CSEs).

6.16.1 See also

Concepts

- *Comparison of pure and impure functions* on page 6-26.

Reference

Compiler Reference:

- *__pure* on page 5-20.

6.17 Comparison of pure and impure functions

The use of the __pure keyword is illustrated in the two sample routines of Table 6-7. Both routines call a function fact() to calculate the sum of n! and n!. fact() 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, fact() is qualified as __pure to indicate to the compiler that it is a pure function.

Table 6-7 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 6-8 shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations of Table 6-7, where the C code for each implementation has been compiled using the option -O2, and inlining has been suppressed.

Table 6-8 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 function foo() in Table 6-8 where fact() is not qualified as __pure, fact() is called twice because the compiler does not know that the function is a candidate for *Common Subexpression Elimination* (CSE). In contrast, in the disassembly of foo() in Table 6-8 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).

6.17.1 See also

Concepts

- *Functions that return the same result when called with the same arguments* on page 6-25.

Reference

Compiler Reference:

- *__pure* on page 5-20.

6.18 Recommendation of postfix syntax when qualifying functions with ARM function modifiers

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 enables 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.

6.18.1 See also

Reference

Compiler Reference:

- *__irq* on page 5-16
- *__pure* on page 5-20
- *__softfp* on page 5-22
- *__svc* on page 5-23
- *__svc_indirect* on page 5-24

- *__value_in_regs* on page 5-26.

6.19 Inline functions

Inline functions offer 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, when compiling with `-Ospace`, 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. When compiling with `-Otime`, the compiler inlines in most cases, but still avoids large code growth. On NEON, calls to non-inline functions from within a loop inhibit vectorization, and require explicit indication that they are to be inlined for vectorization to take place.

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.

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

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.

The linker is able to apply some degree of function inlining to functions that are very short.

6.19.1 See also

Concepts

- *Compiler decisions on function inlining* on page 6-31
- *Restriction on overriding compiler decisions about function inlining* on page 6-36
- *NEON C extensions* on page 4-8
- *Vectorization diagnostics to tune code for improved performance* on page 4-32
- *Nonvectorization of function calls to non-inline functions from within loops* on page 4-30.

Reference

Compiler Reference:

- *--autoinline, --no_autoinline* on page 3-26
- *__forceinline* on page 5-10
- *--forceinline* on page 3-95
- *__inline* on page 5-13.

Linker Reference:

- *--inline, --no_inline* on page 2-85.

6.20 Compiler decisions on function inlining

When function inlining is enabled, the compiler uses a complex decision tree to decide if a function is to be inlined.

The following simplified algorithm is used:

1. If the function is qualified with `__forceinline`, 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, 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, the function is inlined if it is practical to do so.
3. If the optimization level is `-O2` or higher, or `--autoinline` is specified, the compiler automatically inlines functions if it is practical to do so, even if you do not explicitly give a hint that function inlining is wanted.

When deciding if it is practical to inline a function, the compiler takes into account several other criteria, such as:

- 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.

A recursive function is inlined into itself only once, even if `__forceinline` is used.

6.20.1 See also

Concepts

- *Inline functions* on page 6-30
- *Automatic function inlining and static functions* on page 6-33
- *Automatic function inlining and multifile compilation* on page 6-35
- *Restriction on overriding compiler decisions about function inlining* on page 6-36.

Reference

Compiler Reference:

- *--autoinline, --no_autoinline* on page 3-26
- *__forceinline* on page 5-10
- *--forceinline* on page 3-95
- *__inline* on page 5-13
- *--inline, --no_inline* on page 3-122

- -*Onum* on page 3-156
- -*Ospace* on page 3-160
- -*Otime* on page 3-161.

6.21 Automatic function inlining and static functions

At `-O2` and `-O3` levels of optimization, or when `--autoline` is specified, the compiler can automatically inline functions if it is practical and possible to do so, even if the functions are not declared as `__inline` or **`inline`**. This works best for static functions, because if all use of a static function can be inlined, no out-of-line copy is required. 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.

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 being 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 fail to declare functions that are never called from outside a module as **`static`**, 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 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.

6.21.1 See also

Concepts

- *Inline functions* on page 6-30
- *Compiler decisions on function inlining* on page 6-31
- *Automatic function inlining and multifile compilation* on page 6-35
- *Restriction on overriding compiler decisions about function inlining* on page 6-36.

Reference

Compiler Reference:

- *--autoinline, --no_autoinline* on page 3-26
- *-Onum* on page 3-156.

6.22 Inline functions and removal of unused out-of-line functions at link time

The linker cannot remove unused out-of-line functions from an object unless you place the unused out-of-line functions in their own sections using one of the following methods:

- `--split_sections`
- `__attribute__((section("name")))`
- `#pragma arm section [section_type_list]`
- linker feedback.

`--feedback` is typically an easier method of enabling unused function removal.

Reference

Compiler Reference:

- *--feedback=filename* on page 3-93
- *--split_sections* on page 3-193
- *__attribute__((section("name"))) variable attribute* on page 5-77
- *#pragma arm section [section_type_list]* on page 5-88.

6.23 Automatic function inlining and multifile compilation

If you are compiling with the `--multifile` option, which in RVCT 4.0 is 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. `--multifile` is disabled by default in ARM Compiler 4.1 and later, regardless of the optimization level.

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.

Note

`--ltcg` is deprecated.

6.23.1 See also

Concepts

- *Inline functions* on page 6-30
- *Compiler decisions on function inlining* on page 6-31
- *Automatic function inlining and static functions* on page 6-33
- *Restriction on overriding compiler decisions about function inlining* on page 6-36.

Reference

Compiler Reference:

- *--autoinline, --no_autoinline* on page 3-26
- *--inline, --no_inline* on page 3-122
- *--ltcg* on page 3-143
- *--multifile, --no_multifile* on page 3-150
- *-Onum* on page 3-156.

6.24 Restriction on overriding compiler decisions about function inlining

You can enable and disable function inlining, but you cannot override decisions the compiler makes 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 to do so. Even if you use `--forceinline` or `__forceinline`, the compiler only inlines functions if it is possible to do so.

6.24.1 See also

Concepts

- *Compiler decisions on function inlining* on page 6-31.

Reference

Compiler Reference:

- *--autoinline, --no_autoinline* on page 3-26
- *--forceinline* on page 3-95
- *__forceinline* on page 5-10
- *--inline, --no_inline* on page 3-122
- *__inline* on page 5-13.

6.25 Compiler modes and inline functions

Compiler modes affect the behavior of inline functions. See:

- *Inline functions in C++ and C90 mode* on page 6-38
- *Inline functions in C99 mode* on page 6-39
- Inline functions in GNU C90 mode:
  - `http://gcc.gnu.org`.

6.26 Inline functions in 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 translation unit, 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.

6.26.1 See also

Concepts

Using the Linker:

- *Elimination of common groups or sections* on page 5-3.

Reference

Compiler Reference:

- *__inline* on page 5-13
- *--inline, --no_inline* on page 3-122.

6.27 Inline functions in 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**, and not using **extern**. There is also an external definition in one source file. For example:

Example 6-1 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 in example *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++.

6.27.1 See also

Reference

Compiler Reference:

- *__inline* on page 5-13
- *--inline, --no_inline* on page 3-122
- *--ltcg* on page 3-143
- *--multifile, --no_multifile* on page 3-150.

6.28 Inline functions and debugging

The debug view generated for use 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`, `--inline`, `--autoinline` and `--no_autoinline` command-line options.

The debug view can also be adversely affected by retaining both inline and out-of-line copies of a function when out-of-line copies are not required. Functions that are never called from outside a module can be declared as static functions to avoid an unnecessarily complicated debug view.

6.28.1 See also

Concepts

- *Automatic function inlining and static functions* on page 6-33.

Reference

Compiler Reference:

- *--autoinline, --no_autoinline* on page 3-26
- *--forceinline* on page 3-95
- *--inline, --no_inline* on page 3-122
- *__inline* on page 5-13.

Linker Reference:

- *--inline, --no_inline* on page 2-85.

6.29 Types of data alignment

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

- Natural alignment, for example, on a word boundary at 0x1004. 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.

6.29.1 See also

Concepts

- *Advantages of natural data alignment* on page 6-43.
- *Compiler storage of data objects by natural byte alignment* on page 6-44
- *Relevance of natural data alignment at compile time* on page 6-45
- *The __packed qualifier and unaligned data access in C and C++ code* on page 6-47.

Reference

Compiler Reference:

- *#pragma pack(n)* on page 5-107.

6.30 Advantages of natural data alignment

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.

ARM and Thumb processors are designed to efficiently access *naturally aligned* data, that is, doublewords that lie on addresses that are multiples of eight, 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.

6.30.1 See also

Concepts

- *Compiler storage of data objects by natural byte alignment* on page 6-44
- *Relevance of natural data alignment at compile time* on page 6-45.

6.31 Compiler storage of data objects by natural byte alignment

By default, the compiler stores data objects by byte alignment as shown in Table 6-9.

Table 6-9 Compiler storage of data objects by byte alignment

Type	Bytes	Alignment
`char`, `bool`, `_Bool`.	1	Located at any byte address.
`short`, `wchar_t`.	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`, `long double`.	8	Located at an address that is evenly divisible by 8.

6.31.1 See also

Concepts

- *Advantages of natural data alignment* on page 6-43
- *Relevance of natural data alignment at compile time* on page 6-45.

6.32 Relevance of natural data alignment at compile time

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;
};
```

6.32.1 See also

Concepts

- *Advantages of natural data alignment* on page 6-43
- *Compiler storage of data objects by natural byte alignment* on page 6-44
- *Types of data alignment* on page 6-42.

6.33 Unaligned data access in C and C++ code

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 is required when accessing unaligned data in memory, to avoid unexpected results. 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` instruction 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.

6.33.1 See also

Concepts

- *Types of data alignment* on page 6-42
- *The __packed qualifier and unaligned data access in C and C++ code* on page 6-47.

6.34 The __packed qualifier and unaligned data access in C and C++ code

The __packed qualifier sets the alignment of any valid type to 1. This enables objects of packed type to be read or written using unaligned access.

Examples of objects that can be packed include:

- structures
- unions
- pointers.

6.34.1 See also

Concepts

- *Types of data alignment* on page 6-42
- *Unaligned data access in C and C++ code* on page 6-46.

Reference

Compiler Reference:

- *#pragma pack(n)* on page 5-107
- *__packed* on page 5-17.

6.35 Unaligned fields in structures

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

- 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.

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.

6.35.1 See also

Concepts

- *Performance penalty associated with marking whole structures as packed* on page 6-49
- *Comparisons of an unpacked struct, a __packed struct, and a struct with individually __packed fields, and of a __packed struct and a #pragma packed struct* on page 6-52.

Reference

Compiler Reference:

- *#pragma pack(n)* on page 5-107
- *__packed* on page 5-17.

6.36 Performance penalty associated with marking whole structures as packed

Reading from and writing to whole structures qualified with `__packed` requires unaligned accesses and can therefore incur a performance penalty.

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 not naturally aligned.

6.36.1 See also

Concepts

- *Unaligned fields in structures* on page 6-48.

Reference

Compiler Reference:

- *#pragma pack(n)* on page 5-107
- *__packed* on page 5-17.

6.37 Unaligned pointers in C and C++ code

By default, the compiler expects conventional C and C++ pointers to point to naturally aligned words 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, you must specify the `__packed` qualifier when defining the pointer. For example:

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

When a pointer is declared as `__packed`, the 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.

6.37.1 See also

Reference

Compiler Reference:

- *__packed* on page 5-17.

6.38 Unaligned *Load Register* (LDR) instructions generated by the compiler

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.

6.38.1 See also

Reference

Assembler Reference:

- *Memory access instructions* on page 3-9.

Compiler Reference:

- *__packed* on page 5-17.

6.39 Comparisons of an unpacked struct, a __packed struct, and a struct with individually __packed fields, and of a __packed struct and a #pragma packed struct

These comparisons illustrate the differences between the methods of packing structures.

6.39.1 Comparison of an unpacked struct, a __packed struct, and a struct with individually __packed fields

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 6-10.

Table 6-10 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;`

In the first implementation, the **struct** is not packed. In the second implementation, the entire structure is qualified as __packed. In the third implementation, the __packed attribute is removed from the structure and the individual field that is not naturally aligned is declared as __packed.

Table 6-11 shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations of Table 6-10, where the C code for each implementation has been compiled using the option -O2.

Table 6-11 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]`

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. Using -Ospace results in unaligned accesses made through function calls.

In the disassembly of the unpacked **struct** in Table 6-11, 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 6-11 on page 6-52, fields one and three are aligned on their natural size boundaries by default, so the compiler makes aligned accesses. The compiler always carries out aligned word or halfword accesses for fields it can identify as being 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 *ARM Embedded Application Binary Interface* (AEABI) runtime routine __aeabi_uread4 for reading an unsigned word at an unknown alignment to access 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 6-11 on page 6-52, 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. This is 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.

6.39.2 Comparison of a __packed struct and a #pragma packed struct

The differences between a __packed **struct** and a #pragma packed **struct** are illustrated by the two implementations of a **struct** shown in Table 6-12.

Table 6-12 C code for a packed struct and a pragma packed struct

__packed struct	#pragma packed struct

`__packed struct foobar`
`{`
`  char x;`
`  short y[10];`
`};`
 	
`short get_y0(struct foobar *s)`
`{`
`  // Unaligned-capable load`
`  return *s->y;`
`}`
`short *get_y(struct foobar *s)`
`{`
`  return s->y;  // Compile error`
`}`
	`#pragma push`
	`#pragma pack(1)`
	`struct foobar`
	`{`
	`  char x;`
	`  short y[10];`
	`};`
	`#pragma pop`
	`short get_y0(struct foobar *s)`
	`{`
	`  // Unaligned-capable load`
	`  return *s->y;`
	`}`
	`short *get_y(struct foobar *s)`
	`{`
	`  return s->y;  // No error`
	`  // Potentially illegal unaligned load,`
	`  // depending on use of result`
	`}`

In the first implementation, taking the address of a field in a __packed **struct** or a __packed field in a **struct** yields a __packed pointer, and the compiler generates a type error if you try to implicitly cast this to a non-__packed pointer. In the second implementation, in contrast, taking the address of a field in a #pragma packed **struct** does not yield a __packed-qualified pointer. However, the field might not be properly aligned for its type, and dereferencing such an unaligned pointer results in Undefined behavior.

6.39.3 See also

Concepts

- *Unaligned fields in structures* on page 6-48.

Reference

Compiler Reference:

- *-Ospace* on page 3-160
- *-Otime* on page 3-161
- *__packed* on page 5-17
- *__attribute((packed)) type attribute* on page 5-68
- *__attribute__((packed)) variable attribute* on page 5-76
- *#pragma pack(n)* on page 5-107.

Other information

- *Application Binary Interface (ABI) for the ARM Architecture*, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi/index.html`.

6.40 Compiler support for floating-point arithmetic

The compiler provides many features for managing floating-point arithmetic both in hardware and in software. 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.

The ARM processor core does not contain floating-point hardware so 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.
- In hardware, using a hardware *Vector Floating Point* (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.

Code that uses 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.

6.40.1 See also

Concepts

- *Default selection of hardware or software floating-point support* on page 6-57
- *Example of hardware and software support differences for floating-point arithmetic* on page 6-58
- *Half-precision floating-point number format* on page 6-64.

Developing Software for ARM Processors:

- *ARM and Thumb floating-point build options (ARMv6 and earlier)* on page 2-9
- *ARM and Thumb floating-point build options (ARMv7 and later)* on page 2-10.

Using ARM C and C++ Libraries and Floating-Point Support:

- Chapter 4 *Floating-point support*.

Reference

Compiler Reference:

- *#pragma softfp_linkage, #pragma no_softfp_linkage* on page 5-111
- *--cpu=name* on page 3-49
- *--device=name* on page 3-69
- *__fabs intrinsic* on page 5-129
- *--fp16_format=format* on page 3-96
- *--fpmode=model* on page 3-97
- *--fpu=list* on page 3-99

- *--fpu=name* on page 3-100
- *__sqrt intrinsic* on page 5-156
- *C99 floating-point functions* on page 5-182
- *Predefined macros* on page 5-183
- *Hexadecimal floats* on page 4-16
- *Hexadecimal floating-point constants* on page 4-41
- *Limits for floating-point numbers* on page F-5
- *VFP status intrinsic* on page 5-173.

ARM C and C++ Libraries and Floating-Point Support Reference:

- Chapter 3 *Floating-point support*.

Other information

- Institute of Electrical and Electronics Engineers, `http://www.ieee.org`

6.41 Default selection of hardware or software floating-point support

The default target FPU architecture is derived from use of the `--cpu` option.

If the CPU specified with `--cpu` has a VFP coprocessor, the default target FPU architecture is the VFP architecture for that CPU. For example, the option `--cpu ARM1136JF-S` implies the option `--fpu vfpv2`.

If you are building ARM Linux applications using `--arm_linux` or `--arm_linux_paths`, the default is always software floating-point linkage. Even if you specify a CPU that implies an FPU (for example, `--cpu=ARM1136JF-S`), the compiler still defaults to `--fpu=softvfp+vfp`, not `--fpu=vfp`.

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 ARM compilation tools suite of C libraries.

6.41.1 See also

Concepts

- *Compiler support for floating-point arithmetic* on page 6-55
- *Processors and their implicit Floating-Point Units (FPUs)* on page 6-71.

Using ARM C and C++ Libraries and Floating-Point Support:

- Chapter 4 *Floating-point support*.

6.42 Example of hardware and software support differences for floating-point arithmetic

Example 6-2 shows a function implementing floating-point arithmetic in C code.

Example 6-2 Floating-point arithmetic implemented in C code

```
float foo(float num1, float num2)
{
    float temp, temp2;
    temp = num1 + num2;
    temp2 = num2 * num2;
    return temp2 - temp;
}
```

When the C code of Example 6-2 is compiled with the command-line options `--cpu 5TE` and `--fpu softvfp`, the compiler produces machine code with the disassembly shown in Example 6-3. In Example 6-3, floating-point arithmetic is performed in software through calls to library routines such as `__aeabi_fmul`.

Example 6-3 Support for floating-point arithmetic 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
```

However, when the C code of Example 6-2 is compiled with the command-line option `--fpu vfp`, the compiler produces machine code with the disassembly shown in Example 6-4. In Example 6-4, floating-point arithmetic is performed in hardware through floating-point arithmetic instructions such as `VMUL.F32`.

Example 6-4 Support for floating-point arithmetic in hardware

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

6.42.1 See also

Concepts

- *Default selection of hardware or software floating-point support* on page 6-57

- *Compiler support for floating-point arithmetic* on page 6-55.

Reference

Compiler Reference:

- *--cpu=name* on page 3-49
- *--fpu=list* on page 3-99
- *--fpu=name* on page 3-100.

Other information

- *Application Binary Interface (ABI) for the ARM Architecture*, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi/index.html`

6.43 *Vector Floating-Point* (VFP) architectures

VFP architectures provide 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:
  - VFP10 revision 1, as provided by the ARM10200E processor
  - VFP9-S, available as a separately licensable option for the ARM926E, ARM946E and ARM966E processors
  - VFP11, as provided in the ARM1136JF-S, ARM1176JZF-S and ARM11 MPCore processors.
- VFPv3, implemented on ARM architecture v7 and later, for example, the Cortex-A8 processor. 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_fp16, VFPv3 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, an implementation of VFPv3 that provides 16 double-precision registers. It is implemented on ARM architecture v7 processors that support VFP without NEON.
- VFPv3U, an implementation of VFPv3 that can trap floating-point exceptions. It requires software support code.
- VFPv4, implemented on ARM architecture v7 and later, for example, the Cortex-A7 processor. VFPv4 has 32 double-precision registers. VFPv4 adds both half-precision extensions and fused multiply-add instructions to the features of VFPv3.
- VFPv4-D16, an implementation of VFPv4 that provides 16 double-precision registers. It is implemented on ARM architecture v7 processors that support VFP without NEON.
- VFPv4U, an implementation of VFPv4 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.

6.43.1 See also

Concepts

- *Compiler support for floating-point arithmetic* on page 6-55.

Other information

- *ARM Application Note 133 - Using VFP with RVDS*, http://infocenter.arm.com/help/topic/com.arm.doc.dai0133c/index.html

6.44 Limitations on hardware handling of floating-point arithmetic

ARM *Vector Floating-Point* (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.

6.44.1 See also

Concepts

- *Compiler support for floating-point arithmetic* on page 6-55
- *Implementation of Vector Floating-Point (VFP) support code* on page 6-62.

Other information

- Institute of Electrical and Electronics Engineers, `http://www.ieee.org`

6.45 Implementation of *Vector Floating-Point* (VFP) support code

For convenience, an implementation of VFP support code that can be used in your system is provided with your installation of the ARM compilation tools. 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 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 ARM compilation tools distribution at *`install_directory`*`\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 ARM compilation tools distribution at *`install_directory`*`\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 building with `--fpmode=std`
- 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.

6.45.1 See also

Concepts

- *Limitations on hardware handling of floating-point arithmetic* on page 6-61.

Reference

Compiler Reference:

- *--fpmode=model* on page 3-97.

Other information

- *ARM Application Note 133 - Using VFP with RVDS*, `http://infocenter.arm.com/help/topic/com.arm.doc.dai0133c/index.html`

6.46 Compiler and library support for half-precision floating-point numbers

Half-precision floating-point numbers are provided as an optional extension to the *Vector Floating-Point* (VFP)v3 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=`*`format`* compiler command-line option.

The C++ name mangling for the half-precision data type is specified in the C++ generic *Application Binary Interface* (ABI).

6.46.1 See also

Concepts

- *Half-precision floating-point number format* on page 6-64
- *Vector Floating-Point (VFP) architectures* on page 6-60.

Using ARMC and C++ Libraries and Floating-Point Support:

- Chapter 4 *Floating-point support*.

Reference

Compiler Reference:

- *--fp16_format=format* on page 3-96.

Other information

- *C++ ABI for the ARM Architecture*, `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0041-/index.html`

6.47 Half-precision floating-point number format

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 6-1.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	E					T									

Figure 6-1 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 format is as follows:

```
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 format is as follows:

```
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)
```

6.47.1 See also

Concepts

- *Compiler and library support for half-precision floating-point numbers* on page 6-63.

Reference

Compiler Reference:

- *--fp16_format=format* on page 3-96.

Other information

- Institute of Electrical and Electronics Engineers, `http://www.ieee.org`

6.48 Compiler support for 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.

6.48.1 See also

Concepts

- *Compiler support for floating-point arithmetic* on page 6-55
- *Types of floating-point linkage* on page 6-66
- *Compiler options for floating-point linkage and computations* on page 6-67.

6.49 Types of floating-point linkage

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 *Vector Floating-Point* (VFP) coprocessor registers to pass the arguments and return value.

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.

6.49.1 See also

Concepts

- *Compiler support for floating-point computations and linkage* on page 6-65
- *Compiler options for floating-point linkage and computations* on page 6-67
- *Floating-point linkage and computational requirements of compiler options* on page 6-69
- *Processors and their implicit Floating-Point Units (FPUs)* on page 6-71.

Other information

- *Procedure Call Standard for the ARM Architecture ABI*, `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042-/index.html`

6.50 Compiler options for floating-point linkage and computations

By specifying the type of floating-point linkage and floating-point computations you require, you can determine, from Table 6-13, the associated compiler command-line options that are available.

Table 6-13 Compiler options for floating-point linkage and floating-point computations

Linkage		Computations			
Hardware FP linkage	**Software FP linkage**	**Hardware FP coprocessor**	**Software FP library (fplib)**	**Compiler options**	
No	Yes	No	Yes	`--fpu=softvfp`	`--apcs=/softfp`

No	Yes	Yes	No	`--fpu=softvfp+vfpv2`
				`--fpu=softvfp+vfpv3`
				`--fpu=softvfp+vfpv3_fp16`
				`--fpu=softvfp+vfpv3_d16`
				`--fpu=softvfp+vfp3_d16_fp16`
				`--fpu=softvfp+vfpv4`
				`--fpu=softvfp+vfpv4_d16`
				`--fpu=softvfp+fpv4-sp`
					`--apcs=/softfp`

Yes	No	Yes	No	`--fpu=vfp`
				`--fpu=vfpv2`
				`--fpu=vfpv3`
				`--fpu=vfpv3_fp16`
				`--fpu=vfpv3_dp16`
				`--fpu=vfpv3_d16_fp16`
				`--fpu=vpfv4`
				`--fpu=vfpv4_d16`
				`--fpu=fpv4-sp`
					`--apcs=/hardfp`

`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`, `softvfp+vfpv3_d16_fp16`, `softvfp+vfpv4`, `softvfp+vfpv4_d16`, or `softvfp+fpv4-sp`
- 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 softvfp3_d16`, `--fpu softvfpv3_d16_fp16`, `--fpu softvfp+vfpv4`, `softvfp+vfpv4_d16` and `softvfp+fpv4-sp` 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 6-13.) 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.

If you use the `--fpu` option, you need to know the VFP architecture version implemented in the target core. An alternative to `--fpu=softvfp+...` is `--apcs=/softfp`. This gives software linkage with whatever VFP architecture version is implied by `--cpu`. `--apcs=/softfp` and `--apcs=/hardfp` are alternative ways of requesting the integer or floating-point variant of the *Procedure Call Standard for the ARM Architecture* (AAPCS).

To use hardware floating-point linkage when targeting ARM Linux, you must explicitly specify a `--fpu` option that implies hardware floating-point linkage, for example, `--fpu=vfpv3`, or compile with `--apcs=/hardfp`. Hardware floating-point linkage is not supported by the ARM Linux ABI. The compiler issues a warning to indicate this.

6.50.1 See also

Concepts

Using the Compiler:

- *Compiler support for floating-point computations and linkage* on page 6-65
- *Types of floating-point linkage* on page 6-66
- *Floating-point linkage and computational requirements of compiler options* on page 6-69
- *Processors and their implicit Floating-Point Units (FPUs)* on page 6-71.

Reference

Compiler Reference:

- *--apcs=qualifer...qualifier* on page 3-11
- *--fpu=name* on page 3-100
- *--library_interface=lib* on page 3-128
- *__softfp* on page 5-22
- *#pragma softfp_linkage, #pragma no_softfp_linkage* on page 5-111.

Other information

- *ARM GNU/Linux Application Binary Interface Supplement*, Version 1.0, `http://www.codesourcery.com/sgpp/lite/arm/arm_gnu_linux_abi.pdf`

6.51 Floating-point linkage and computational requirements of compiler options

There are various valid combinations of FPU options and processors. Table 6-14 sets out the FPU options, and their capabilities and requirements.

Table 6-14 FPU-option capabilities and requirements

FPU name	Hardware FP linkage	d0-d15 registers	d16-d31 registers	VFP instructions	Half precision	Single precision	Double precision
softvfp	No	No	No	No	No	No	No
softvfp+vfpv2	No	Yes	No	Yes	No	Yes	Yes
softvfp+vfpv3	No	Yes	Yes	Yes	No	Yes	Yes
softvfp+vfpv3_fp16	No	Yes	Yes	Yes	Yes	Yes	Yes
softvfp+vfpv3_d16	No	Yes	No	Yes	No	Yes	Yes
softvfp+vfpv3_d16_fp16	No	Yes	No	Yes	Yes	Yes	Yes
softvfp+vfpv3_sp_d16	No	Yes	No	Yes	Yes	Yes	No
softvfp+vfpv4	No	Yes	Yes	Yes	Yes	Yes	Yes
softvfp+vfpv4_d16	No	Yes	No	Yes	Yes	Yes	Yes
softvfp+vfpv4_sp_d16	No	Yes	No	Yes	Yes	Yes	No
softvfp+fpv4-sp	No	Yes	No	Yes	Yes	Yes	No
vfp	Yes	Yes	No	Yes	No	Yes	Yes
vfpv2	Yes	Yes	No	Yes	No	Yes	Yes
vfpv3	Yes	Yes	Yes	Yes	No	Yes	Yes
vfpv3_fp16	Yes	Yes	Yes	Yes	Yes	Yes	Yes
vfpv3_d16	Yes	Yes	No	Yes	No	Yes	Yes
vfpv3_d16_fp16	Yes	Yes	No	Yes	Yes	Yes	Yes
vfpv3_sp_d16	Yes	Yes	No	Yes	Yes	Yes	No
vfpv4	Yes	Yes	Yes	Yes	Yes	Yes	Yes
vfpv4_d16	Yes	Yes	No	Yes	Yes	Yes	Yes
vfpv4_sp_d16	Yes	Yes	No	Yes	Yes	Yes	No
fpv4-sp	Yes	Yes	No	Yes	Yes	Yes	No

Note

You can specify the floating-point linkage, independently of the VFP architecture, with `--apcs`.

6.51.1 See also

Concepts

Using the Compiler:

- *Compiler support for floating-point arithmetic* on page 6-55
- *Vector Floating-Point (VFP) architectures* on page 6-60

- *Types of floating-point linkage* on page 6-66
- *Compiler options for floating-point linkage and computations* on page 6-67
- *Processors and their implicit Floating-Point Units (FPUs)* on page 6-71.

References

Compiler Reference:

- *--apcs=qualifer...qualifier* on page 3-11
- *--fpu=name* on page 3-100.

6.52 Processors and their implicit *Floating-Point Units* (FPUs)

Not every ARM processor has an FPU, but every one has an implicit `--fpu` option. Table 6-15 lists the implicit `--fpu` option for each processor `--cpu` option.

Table 6-15 Implicit FPUs of CPUs

CPU name	**FPU** name
ARM processors designed by ARM Limited	
ARM7EJ-S	softvfp
ARM7TDM	softvfp
ARM7TDMI	softvfp
ARM7TDMI-S	softvfp
ARM7TM	softvfp
ARM7TM-S	softvfp
ARM710T	softvfp
ARM720T	softvfp
ARM740T	softvfp
ARM810	softvfp
ARM9E-S	softvfp
ARM9EJ-S	softvfp
ARM9TDMI	softvfp
ARM920T	softvfp
ARM922T	softvfp
ARM926EJ-S	softvfp
ARM940T	softvfp
ARM946E-S	softvfp
ARM966E-S	softvfp
ARM968E-S	softvfp
ARM1020E	softvfp
ARM1022E	softvfp
ARM1026EJ-S	softvfp
ARM1136J-S	softvfp
ARM1136J-S-rev1	softvfp
ARM1136JF-S	vfpv2
ARM1136JF-S-rev1	vfpv2
ARM1156T2-S	softvfp
ARM1156T2F-S	vfpv2

Table 6-15 Implicit FPUs of CPUs (continued)

CPU name	**FPU** name
ARM1176JZ-S	softvfp
ARM1176JZF-S	vfpv2
Cortex-A5	softvfp
Cortex-A5.vfp	vfpv4_d16
Cortex-A5.neon	vfpv4
Cortex-A7	vfpv4
Cortex-A7.no_neon	vfpv4_d16
Cortex-A7.no_neon.no_vfp	softvfp
Cortex-A8	vfpv3
Cortex-A8.no_neon	softvfp
Cortex-A8NoNeon	softvfp
Cortex-A9	vfpv3_fp16
Cortex-A9.no_neon	vfpv3_d16_fp16
Cortex-A9.no_neon.no_vfp	softvfp
Cortex-A15	vfpv4
Cortex-A15.no_neon	vfpv4_d16
Cortex-A15.no_neon.no_vfp	softvfp
Cortex-M0	softvfp
Cortex-M0plus	softvfp
Cortex-M1	softvfp
Cortex-M1.os_extension	softvfp
Cortex-M1.no_os_extension	softvfp
Cortex-M3	softvfp
Cortex-M3-rev0	softvfp
Cortex-M4	softvfp
Cortex-M4.fp	vfpv4-spvfpv4_sp_d16
Cortex-R4	softvfp
Cortex-R4F	vfpv3_d16
Cortex-R5	softvfp
Cortex-R5-rev1	softvfp
Cortex-R5F	vfpv3_d16
Cortex-R5F-rev1	vfpv3_d16
Cortex-R5F-rev1.sp	vfpv3_sp_d16

Table 6-15 Implicit FPUs of CPUs (continued)

CPU name	**FPU** name
Cortex-R7	vfpv3_d16_fp16
Cortex-R7.no_vfp	softvfp
MPCore	vfpv2
MPCore.no_vfp	softvfp
MPCoreNoVFP	softvfp
SC000	softvfp
SC300	softvfp
ARM processors designed by ARM licensees	
88FR101	softvfp
88FR101.hw_divide	softvfp
88FR111	softvfp
88FR111.no_hw_divide	softvfp
88FR121	softvfp
88FR121.hw_divide	softvfp
88FR131	softvfp
88FR131.hw_divide	softvfp
88FR301	softvfp
88FR301.hw_divide	softvfp
88FR321	softvfp
88FR321.hw_divide	softvfp
88FR331	softvfp
88FR331.hw_divide	softvfp
PJ4	vfpv3_d16
PJ4.no_vfp	softvfp
QSP	vfpv3_fp16
QSP.no_neon	vfpv3_fp16
QSP.no_neon.no-vfp	softvfp
SA-110	softvfp
SA-1100	softvfp

Note

You can:

- specify a different FPU with `--fpu`
- specify the floating-point linkage, independently of the FPU architecture, with `--apcs`

- display the complete expanded command-line, including the FPU, with `--echo`.

6.52.1 See also

Concepts

Using the Compiler:

- *Compiler support for floating-point arithmetic* on page 6-55
- *Default selection of hardware or software floating-point support* on page 6-57
- *Vector Floating-Point (VFP) architectures* on page 6-60
- *Compiler options for floating-point linkage and computations* on page 6-67
- *Floating-point linkage and computational requirements of compiler options* on page 6-69.

References

Compiler Reference:

- *--apcs=qualifer...qualifier* on page 3-11
- *--echo* on page 3-83
- *--fpu=name* on page 3-100.

6.53 Integer division-by-zero errors in C code

You can trap and identify integer division-by-zero errors with the appropriate C library helper functions, `__aeabi_idiv0()` and `__rt_raise()`.

6.53.1 See also

Tasks

- *Examining parameters when integer division-by-zero errors occur in C code* on page 6-79.

Concepts

- *About trapping integer division-by-zero errors with __aeabi_idiv0()* on page 6-76
- *About trapping integer division-by-zero errors with __rt_raise()* on page 6-77
- *Identification of integer division-by-zero errors in C code* on page 6-78.

Other information

- *Run-time ABI for the ARM Architecture*, `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0043-/index.html`.

6.54 About trapping integer division-by-zero errors with __aeabi_idiv0()

You can trap integer division-by-zero errors with 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 only have to place a breakpoint on `__aeabi_idiv0()`.

The library provides two implementations of `__aeabi_idiv0()`. The default one does nothing, so if division by zero is detected, the division function returns zero. However, if you use signal handling, an alternative implementation is selected that calls `__rt_raise(SIGFPE, DIVBYZERO)`.

If you provide your own version of `__aeabi_idiv0()`, then the division functions call this function. The function prototype for `__aeabi_idiv0()` is:

```
int __aeabi_idiv0(void);
```

If `__aeabi_idiv0()` returns a value, that value is used as the quotient returned by the division function.

6.54.1 See also

Concepts

- *Integer division-by-zero errors in C code* on page 6-75
- *About trapping integer division-by-zero errors with __rt_raise()* on page 6-77.

Other information

- *Run-time ABI for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0043-/index.html.

6.55 About trapping integer division-by-zero errors with `__rt_raise()`

By default, integer division by zero returns zero. If you want to intercept division by zero, you can re-implement the C library helper function `__rt_raise()`. The function prototype for `__rt_raise()` is:

```
void __rt_raise(int signal, int type);
```

If you re-implement `__rt_raise()`, then the library automatically provides the signal-handling library version of `__aeabi_idiv0()`, which calls `__rt_raise()`, then that library version of `__aeabi_idiv0()` is included in the final image.

In that case, when a divide-by-zero error occurs, `__aeabi_idiv0()` calls `__rt_raise(SIGFPE, DIVBYZERO)`. Therefore, if you re-implement `__rt_raise()`, you must check `(signal == SIGFPE) && (type == DIVBYZERO)` to determine if division by zero has occurred.

6.55.1 See also

Concepts

- *Integer division-by-zero errors in C code* on page 6-75
- *About trapping integer division-by-zero errors with __aeabi_idiv0()* on page 6-76.

Using ARM C and C++ Libraries and Floating-Point Support:

- *Integer and floating-point compiler functions and building an application without the C library* on page 2-45
- *Customized C library startup code and access to C library functions* on page 2-48
- *Modification of C library functions for error signaling, error handling, and program exit* on page 2-80
- *ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments* on page 2-110.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__rt_raise()* on page 2-35.

Other information

- *Run-time ABI for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0043-/index.html.

6.56 Identification of integer division-by-zero errors in C 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. The offending line in the source code can be identified by looking up the line of C code in the debugger at the address given by `LR`.

6.56.1 See also

Concepts

- *Integer division-by-zero errors in C code* on page 6-75.

Other information

- *Run-time ABI for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0043-/index.html.

6.57 Examining parameters when integer division-by-zero errors occur in C code

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.

To examine parameters when integer division-by-zero occurs:

1. Prefix `__aeabi_idiv0()` with `$Super$$` to identify the original unpatched function `__aeabi_idiv0()`.
2. Use `__aeabi_idiv0()` prefixed with `$Super$$` to call the original function directly.
3. Prefix `__aeabi_idiv0()` with `$Sub$$` to identify the new function to be called in place of the original version of `__aeabi_idiv0()`.
4. Use `__aeabi_idiv0()` prefixed with `$Sub$$` to add processing before or after the original function `__aeabi_idiv0()`.

6.57.1 Example

Example 6-5 illustrates the use of the `$Super$$` and `$Sub$$` mechanism to intercept `__aeabi_div0`.

Example 6-5 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();
}
```

6.57.2 See also

Concepts

- *Integer division-by-zero errors in C code* on page 6-75.

Using the Linker:

- *Using $Super$$ and $Sub$$ to patch symbol definitions* on page 7-29.

Other information

- *Run-time ABI for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0043-/index.html.

6.58 Software floating-point division-by-zero errors in C code

Floating-point division-by-zero errors in software can be trapped and identified using a combination of intrinsics and C library helper functions. See the following topics:

- *About trapping software floating-point division-by-zero errors* on page 6-81
- *Identification of software floating-point division-by-zero errors* on page 6-82
- *Software floating-point division-by-zero debugging* on page 6-84.

6.59 About trapping software floating-point division-by-zero errors

Software floating-point division-by-zero errors can be trapped with the following 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 6-6.

Example 6-6 Trapped 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;
}
```

6.59.1 See also

Concepts

- *Software floating-point division-by-zero errors in C code* on page 6-80.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__ieee_status()* on page 3-8.

6.60 Identification of software floating-point division-by-zero errors

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 the application code that contains the arithmetic operation that resulted in the exception, a breakpoint can be placed on the function `_fp_trapveneer()` and `LR` can be inspected.

For example, suppose the C code of Example 6-7 is compiled from the command line using the string:

```
armcc --fpmode ieee_full
```

When the assembly language code produced by the compiler is disassembled, the debugger produces the output shown in Example 6-8.

Example 6-7 Trapped 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;
}
```

Example 6-8 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.

6.60.1 See also

Concepts

- *Software floating-point division-by-zero errors in C code* on page 6-80.

6.61 Software floating-point division-by-zero debugging

Parameters for postmortem debugging can be saved by intercepting `_fp_trapveneer()`. The `$Super$$` and `$Sub$$` mechanism can be used to intervene in all calls to `_fp_trapveneer()`. 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
```

6.61.1 See also

Tasks

- *Examining parameters when integer division-by-zero errors occur in C code* on page 6-79

Using the Linker:

- *Using $Super$$ and $Sub$$ to patch symbol definitions* on page 7-29.

Concepts

- *Software floating-point division-by-zero errors in C code* on page 6-80.

6.62 New language features of C99

The 1999 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 statements.
- Some entirely new features, for example complex numbers, restricted pointers and designated initializers.
- New keywords and identifiers.
- Extended syntax for the existing C90 language.

A selection of new features in C99 that might be of interest to developers using them for the first time are documented.

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.

Some examples of special cases where the language specified by the C90 standard is not a subset of C++ include support for // comments and merging of the typedef and structure tag namespaces. For example, in C90 the following code expands to `x = a / b - c;` because `/* hello world */` is deleted, but in C++ and C99 it expands to `x = a - c;` because everything from // to the end of the line is deleted:

```
x = a //* hello world */ b
  - c;
```

The following code demonstrates how typedef and the structure tag are treated differently between C (90 and 99) and C++ because of their merged namespaces:

```
typedef int a;
{
  struct a { int x, y; };
  printf("%d\n", sizeof(a));
}
```

In C 90 and C99, this code defines two types with separate names whereby `a` is a typedef for `int` and `struct a` is a structure type containing two integer data types. `sizeof(a)` evaluates to `sizeof(int)`.

In C++, a structure type can be addressed using only its tag. This means that when the definition of `struct a` is in scope, the name `a` used on its own refers to the structure type rather than the typedef, so in C++ `sizeof(a)` is greater than `sizeof(int)`.

6.62.1 See also

Concepts

- *// comments* on page 4-9
- *Compound literals in C99* on page 6-89
- *Designated initializers in C99* on page 6-90
- *Hexadecimal floating-point numbers in C99* on page 6-91
- *Flexible array members in C99* on page 6-92
- *__func__ predefined identifier in C99* on page 6-93
- *inline functions in C99* on page 6-94
- *long long data type in C99 and C90* on page 6-95
- *Macros with a variable number of arguments in C99* on page 6-96
- *Mixed declarations and statements in C99* on page 6-97
- *New block scopes for selection and iteration statements in C99* on page 6-98
- *_Pragma preprocessing operator in C99* on page 6-99
- *Restricted pointers in C99* on page 6-100
- *Complex numbers in C99* on page 6-102.

6.63 New library features of C99

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>`.
- New libraries, and new macros and functions for existing C90 libraries.

A selection of new features in C99 that might be of interest to developers using them for the first time are documented.

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.

Many library features that are new to C99 are available in C90 and C++. Some require macros such as `USE_C99_ALL` or `USE_C99_MATH` to be defined before the `#include`.

6.63.1 See also

Concepts

- *Additional <math.h> library functions in C99* on page 6-101
- *Complex numbers in C99* on page 6-102
- *Boolean type and <stdbool.h> in C99* on page 6-103
- *Extended integer types and functions in <inttypes.h> and <stdint.h> in C99* on page 6-104
- *<fenv.h> floating-point environment access in C99* on page 6-105
- *<stdio.h> snprintf family of functions in C99* on page 6-106
- *<tgmath.h> type-generic math macros in C99* on page 6-107
- *<wchar.h> wide character I/O functions in C99* on page 6-108.

6.64 // comments in C99 and C90

In C99 you can use // to indicate the start of a one-line comment, like in C++.

In C90 mode you can use // comments providing you do not specify `--strict`.

6.64.1 See also

Concepts

- *New language features of C99* on page 6-85.

Compiler Reference:

- *--strict, --no_strict* on page 3-194.

Reference

Compiler Reference:

- *// comments* on page 4-9.

6.65 Compound literals in C99

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};
```

Note

`int *y = (int []) {1, 2, 3};` is accepted by the compiler, but `int y[] = (int []) {1, 2, 3};` is not accepted as a high-level (global) initialization.

In the following example source code, the compound literals are:

- `(struct T) { 43, "world"}`
- `&(struct T) {.b = "hello", .a = 47}`
- `&(struct T) {43, "hello"}`
- `(int[]){1, 2, 3}`

```
struct T
{
   int a;
   char *b;
} t2;

void g(const struct T *t);

void f()
{
   int x[10];

   ...

   t2 = (struct T) {43, "world"};
   g(&(struct T) {.b = "hello", .a = 47});
   g(&(struct T) {43, "bye"});
   memcpy(x, (int[]){1, 2, 3}, 3 * sizeof(int));
}
```

6.65.1 See also

Concepts

- *New language features of C99* on page 6-85.

6.66 Designated initializers in C99

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.

6.66.1 See also

Concepts

- *New language features of C99* on page 6-85.

6.67 Hexadecimal floating-point numbers in C99

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`.

C99 also adds `%a` and `%A` format for `printf()`.

6.67.1 See also

Concepts

- *New language features of C99* on page 6-85.

6.68 Flexible array members in C99

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));
}
```

6.68.1 See also

Concepts

- *New language features of C99* on page 6-85.

6.69 __func__ predefined identifier in C99

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'.
```

6.69.1 See also

Concepts

- *New language features of C99* on page 6-85.

Reference

- *Built-in function name variables* on page 5-189.

6.70 `inline` functions in C99

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.

Note

The **`__inline`** keyword is available in C90.

Note

The semantics of **`inline`** in C99 are different to the semantics of **`inline`** in Standard C++.

6.70.1 See also

Concepts

- *Inline functions* on page 6-30
- *New language features of C99* on page 6-85.

6.71 `long long` data type in C99 and C90

C99 supports the integral data type **`long long`**. This type is 64 bits wide in the ARM compilation tools. For example:

```
long long int j = 25902068371200;               // length of light day, meters
unsigned long long int i = 946073047258080000LL; // length of light year, meters
```

`long long` is also available in C90 when not using `--strict`.

`__int64` is a synonym for **`long long`**. `__int64` is always available.

6.71.1 See also

Concepts

- *New language features of C99* on page 6-85.

Reference

Compiler Reference:

- *long long* on page 4-14.

6.72 Macros with a variable number of arguments in C99

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);
}
```

6.72.1 See also

Concepts

- *New language features of C99* on page 6-85.

6.73 Mixed declarations and statements in C99

C99 enables you to mix declarations and statements within compound statements, like in C++. For example:

```
void foo(float i)
{
    i = (i > 0) ? -i : i;
    float j = sqrt(i);    // illegal in C90
}
```

6.73.1 See also

Concepts

- *New language features of C99* on page 6-85.

6.74 New block scopes for selection and iteration statements in C99

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.

6.74.1 See also

Concepts

- *New language features of C99* on page 6-85.

6.75 _Pragma preprocessing operator in C99

C90 does not permit a #pragma directive to be produced as the result of a macro expansion. However, the C99 _Pragma operator enables you to embed a preprocessor macro in a pragma directive, and _Pragma is permitted in C90 if --strict is not specified. 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"
```

6.75.1 See also

Concepts

- *New language features of C99* on page 6-85.

6.76 Restricted pointers in C99

The C99 keyword **restrict** is an indication to the compiler 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 must not, 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.

It is your responsibility to ensure that **restrict**-qualified pointers do not point to overlapping regions of memory.

__restrict, permitted in C90 and C++, is a synonym for **restrict**.

`--restrict` enables **restrict** to be used in C90 and C++.

6.76.1 See also

Concepts

- *New language features of C99* on page 6-85.

Reference

Compiler Reference:

- *--restrict, --no_restrict* on page 3-183.

6.77 Additional <math.h> library functions in C99

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.

6.77.1 See also

Concepts

- *New library features of C99* on page 6-87.

Other information

- Institute of Electrical and Electronics Engineers, http://www.ieee.org

6.78 Complex numbers in C99

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`**.

6.78.1 See also

Concepts

- *New library features of C99* on page 6-87.

6.79 Boolean type and <stdbool.h> in C99

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++.

6.79.1 See also

Concepts

- *New library features of C99* on page 6-87.

6.80 Extended integer types and functions in <inttypes.h> and <stdint.h> in C99

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
```

These header files are also available in C90 and C++.

6.80.1 See also

Concepts

- *New library features of C99* on page 6-87.

6.81 <fenv.h> floating-point environment access in C99

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
```

6.81.1 See also

Concepts

- *New library features of C99* on page 6-87.

Other information

- Institute of Electrical and Electronics Engineers, http://www.ieee.org

6.82 <stdio.h> snprintf family of functions in C99

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.

6.82.1 See also

Concepts

- *New library features of C99* on page 6-87.

6.83 <tgmath.h> type-generic math macros in C99

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>.

6.83.1 See also

Concepts

- *New library features of C99* on page 6-87.

6.84 <wchar.h> wide character I/O functions in C99

Wide character I/O functions have been incorporated into 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`.

6.84.1 See also

Concepts

- *New library features of C99* on page 6-87.

Other information

- Section 7.24, *ISO/IEC9899:TC2*.

6.85 How to prevent uninitialized data from being initialized to zero

The ANSI C specification states that static data that is not explicitly initialized, is to be initialized to zero. Therefore, by default, the compiler puts both zero-initialized and uninitialized data into the same ZI data section, which is populated with zeroes at runtime by the C library initialization code.

You can prevent uninitialized data from being initialized to zero by placing that data in a different section. This can be achieved using `#pragma arm section`, or with the GNU compiler extension `__attribute__((section("name")))`.

Example 6-9 Retaining uninitialized data using `#pragma arm section`

```
#pragma arm section zidata = “non_initialized”
int i, j; // uninitialized data in non_initialized section (without the pragma,
             would be in .bss section by default)
#pragma arm section zidata // back to default (.bss section)
int k = 0, l = 0; // zero-initialized data in .bss section
```

The `non_initialized` section is then placed into its own `UNINIT` execution region:

```
LOAD_1 0x0
{
  EXEC_1 +0
  {
    * (+RO)
    * (+RW)
    * (+ZI) ; ZI data gets initialized to zero
  }
  EXEC_2 +0 UNINIT
  {
    * (non_init) ; ZI data does not get initialized to zero
  }
}
```

6.85.1 See also

Reference

Compiler Reference:

- *#pragma arm section [section_type_list]* on page 5-88
- *__attribute__((section("name"))) variable attribute* on page 5-77
- *--gnu* on page 3-107.

Linker Reference:

- *Execution region attributes* on page 4-12.

Chapter 7
Compiler Diagnostic Messages

The compiler issues messages about potential portability problems and other hazards. It is possible to:

- Turn off specific messages. For example, warnings can be turned off 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 turn off messages.
- Change the severity of specific messages.

The following topics describe the format of compiler diagnostic messages and how to control the output during compilation:

7.1 About compiler diagnostic messages

Severity of diagnostics

- *Severity of compiler diagnostic messages* on page 7-3
- *Options that change the severity of compiler diagnostic messages* on page 7-4.

Diagnostics output control

Compiler Reference:

- *--brief_diagnostics, --no_brief_diagnostics* on page 3-29
- *--diag_style={arm|ide|gnu}* on page 3-72
- *--diag_suppress=tag[,tag,...]* on page 3-73
- *--errors=filename* on page 3-86
- *--remarks* on page 3-181
- *--wrap_diagnostics, --no_wrap_diagnostics* on page 3-228.

Prefix letters in diagnostic messages

- *Prefix letters in compiler diagnostic messages* on page 7-5.

Suppression of warning messages

Compiler Reference:

- *-W* on page 3-220.

Exit status codes and termination messages

- *Compiler exit status codes and termination messages* on page 7-6.

Compiler data flow warnings

- *Compiler data flow warnings* on page 7-7.

7.2 Severity of compiler diagnostic messages

Diagnostic messages have an associated *severity*. See Table 7-1.

Table 7-1 Severity of diagnostic messages

Severity	Description
Internal fault	Internal faults indicate an internal problem with the compiler. Contact your supplier with feedback.
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, 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.

7.2.1 See also

Tasks

- Chapter 1 *Conventions and Feedback*.

Concepts

- *About compiler diagnostic messages* on page 7-2.

7.3 Options that change the severity of compiler 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, or tags, to Error severity.

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

Sets the diagnostic messages that have the specified tag, or tags, to Remark severity.

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

Sets the diagnostic messages that have the specified tag, or tags, 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 ...
```

Only errors with a suffix of `-D` following the error number can be downgraded by changing them into warnings or remarks.

Note

These options also have pragma equivalents.

The following diagnostic messages can be changed:

- Messages with the number format `#nnnn-D`.
- Warning messages with the number format `Cnnnn W`.

It is also possible to apply changes to optimization messages as a group. For example, `--diag_warning=optimizations`. By default, optimization messages are remarks.

7.3.1 See also

Concepts

- *About compiler diagnostic messages* on page 7-2.

Reference

Compiler Reference:

- *Pragmas recognized by the compiler* on page 5-23.

7.4 Prefix letters in compiler diagnostic messages

The compilation tools automatically insert an identification letter to diagnostic messages. See Table 7-2. Using these prefix letters enables the tools to use overlapping message ranges.

Table 7-2 Identifying diagnostic messages

Prefix letter	Tool
C	armcc
A	armasm
L	armlink or armar
Q	fromelf

The following rules apply:

- All of the compilation 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.

Therefore, 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.

7.4.1 See also

Concepts

- *About compiler diagnostic messages* on page 7-2.

7.5 Compiler 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.

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.

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.

7.5.1 See also

Concepts

- *About compiler diagnostic messages* on page 7-2
- *Severity of compiler diagnostic messages* on page 7-3.

7.6 Compiler 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 *RealView Compiler Tools* (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.

7.6.1 See also

Concepts

- *About compiler diagnostic messages* on page 7-2.

Chapter 8
Using the Inline and Embedded Assemblers of the ARM Compiler

The following topics describe the optimizing inline assembler and non-optimizing embedded assembler of the ARM compiler, `armcc`.

Note

Using intrinsics is generally preferable to using inline or embedded assembly language. See *Compiler intrinsics* on page 5-3.

8.1 Compiler support for inline assembly language

The compiler provides an inline assembler that enables you to write optimized assembly language routines, and to access features of the target processor not available from C or C++.

See the following topics:

- *Inline assembler support in the compiler* on page 8-5
- *Restrictions on inline assembler support in the compiler* on page 8-6
- *Inline assembly language syntax with the __asm keyword in C and C++* on page 8-7
- *Inline assembly language syntax with the asm keyword in C++* on page 8-8
- *Inline assembler rules for compiler keywords __asm and asm* on page 8-9
- *Restrictions on inline assembly operations in C and C++ code* on page 8-11
- *Inline assembler and register access in C and C++ code* on page 8-18
- *Inline assembler and the # constant expression specifier in C and C++ code* on page 8-20
- *Inline assembler and instruction expansion in C and C++ code* on page 8-21
- *Inline assembler effect on processor condition flags in C and C++ code* on page 8-24
- *Inline assembler operands in C and C++ code* on page 8-25
- *Inline assembler function calls and branches in C and C++ code* on page 8-29
- *Inline assembler branches and labels in C and C++ code* on page 8-34.

For information on how to use register variables as an alternative to some uses of inline assembly language:

- *Compiler Reference*:
  - *Named register variables* on page 5-176.

For information on how to use the inline assembler in C and C++ source code, and restrictions on inline assembly language:

- *Developing Software for ARM Processors*:
  - Chapter 4 *Mixing C, C++, and Assembly Language*.

For more information on writing assembly language for ARM processors:

- *Using the Assembler*.

8.2 Inline assembler support in the compiler

The inline assembler supports ARM assembly language for all architectures, and Thumb assembly language in ARMv6T2, ARMv6M, and ARMv7.

For ARMv7, the inline assembler supports:

- most ARM instructions
- most Thumb instructions.

For ARMv6T2, the inline assembler supports most Thumb instructions.

For ARMv6, the inline assembler supports most ARM instructions, including the complete set of ARMv6 *Single Instruction Multiple Data* (SIMD) instructions.

For ARMv5, the inline assembler supports most ARM instructions, including generic coprocessor instructions.

For ARMv4, the inline assembler supports most ARM instructions, including generic coprocessor instructions.

VFPv2 instructions are supported in the inline assembler.

8.2.1 See also

Concepts

- *Compiler support for inline assembly language* on page 8-4
- *Restrictions on inline assembler support in the compiler* on page 8-6
- *Inline assembly language syntax with the __asm keyword in C and C++* on page 8-7
- *Embedded assembler support in the compiler* on page 8-36
- *Differences in compiler support of inline and embedded assembly code* on page 8-58.

8.3 Restrictions on inline assembler support in the compiler

The inline assembler in the compiler does not support:

- Thumb assembly language in processors without Thumb-2 technology
- VFP instructions that were added in VFPv3 or higher
- NEON instructions
- the ARMv6 `SETEND` instruction and some of the system extensions
- ARMv5 `BX`, `BLX`, and `BXJ` instructions.

8.3.1 See also

Concepts

- *Inline assembler support in the compiler* on page 8-5
- *Restrictions on inline assembly operations in C and C++ code* on page 8-11
- *Differences in compiler support of inline and embedded assembly code* on page 8-58.

Reference

Assembler Reference:

- *LDM and STM* on page 3-70.

8.4 Inline assembly language syntax with the __asm keyword in C and C++

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]");
  __asm{instruction[;instruction]}
  ```

  You cannot include comments.

- Using multiple adjacent strings, for example:

  ```
  __asm("ADD x, x, #1\n"
        "MOV y, x\n");
  ```

  This enables you to use macros to generate inline assembly, for example:

  ```
  #define ADDLSL(x, y, shift) __asm ("ADD " #x ", " #y ", LSL " #shift)
  ```

- On multiple lines, for example:

  ```
  __asm
  {
      ...
      instruction
      ...
  }
  ```

  You can use C or C++ comments anywhere in an inline assembly language block.

You can use an `__asm` statement wherever a statement is expected.

8.4.1 See also

Concepts

- *Inline assembler rules for compiler keywords __asm and asm* on page 8-9
- *Inline assembler support in the compiler* on page 8-5.

8.5 Inline assembly language syntax with the asm keyword in C++

When compiling C++, the 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]");
  asm{instruction[;instruction]}
  ```

  You cannot include comments.

- Using multiple adjacent strings, for example:

  ```
  asm("ADD x, x, #1\n"
        "MOV y, x\n");
  ```

  This enables you to use macros to generate inline assembly, for example:

  ```
  #define ADDLSL(x, y, shift) asm ("ADD " #x ", " #y ", LSL " #shift)
  ```

- On multiple lines, for example:

  ```
  asm
  {
      ...
      instruction
      ...
  }
  ```

  You can use C or C++ comments anywhere in an inline assembly language block.

You can use an `asm` statement wherever a statement is expected.

8.5.1 See also

Concepts

- *Inline assembler rules for compiler keywords __asm and asm* on page 8-9
- *Inline assembler support in the compiler* on page 8-5.

8.6 Inline assembler rules for compiler keywords __asm and asm

The following rules apply to the **__asm** and **asm** keywords:

- Multiple instructions on the same line must be separated with a semicolon (;).
- If an instruction requires more than one line, line continuation must be specified with the backslash character (\).
- For the multiple line format, C and C++ comments are permitted anywhere in the inline assembly language block. However, comments cannot be embedded 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)
}
```

- Labels must be followed by a colon, :, like C and C++ labels.
- 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. If the register is not declared as a C or C++ variable, the compiler generates a warning.
- Registers must not be saved and restored in inline assembler. The compiler does this for you. Also, the inline assembler does not provide direct access to the physical registers. However, indirect access is provided through variables that act as virtual registers.

  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;
}
```

8.6.1 See also

Concepts

- *Inline assembler support in the compiler* on page 8-5
- *Restrictions on inline assembler support in the compiler* on page 8-6
- *Restrictions on inline assembly operations in C and C++ code* on page 8-11.

8.7 Restrictions on inline assembly operations in C and C++ code

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.

See the following restrictions:

- *Inline assembler register restrictions in C and C++ code* on page 8-12
- *Inline assembler processor mode restrictions in C and C++ code* on page 8-13
- *Inline assembler Thumb instruction set restrictions in C and C++ code* on page 8-14
- *Inline assembler Vector Floating-Point (VFP) restrictions in C and C++ code* on page 8-15
- *Inline assembler instruction restrictions in C and C++ code* on page 8-16
- *Miscellaneous inline assembler restrictions in C and C++ code* on page 8-17
- *Restrictions on inline assembler support in the compiler* on page 8-6.

8.8 Inline assembler register restrictions in C and C++ code

Registers such as `r0-r3`, `sp`, `lr`, and the NZCV flags in the `CPSR` must be used with caution. If C or C++ expressions are used, these might be used as temporary registers and NZCV flags might be corrupted by the compiler when evaluating the expression.

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, the intrinsics `__current_pc`, `__current_sp`, and `__return_address` can be used to read these registers.

8.8.1 See also

Concepts

- *Inline assembler and register access in C and C++ code* on page 8-18
- *Restrictions on inline assembly operations in C and C++ code* on page 8-11.

Reference

Compiler Reference:

- *__current_pc intrinsic* on page 5-122
- *__current_sp intrinsic* on page 5-123
- *__return_address intrinsic* on page 5-150.

8.9 Inline assembler processor mode restrictions in C and C++ code

Caution

It is strongly recommended that you do not change processor modes or modify coprocessor states in inline assembler. The compiler does not recognize such changes.

Instead of attempting to change processor modes or coprocessor states from within inline assembler, see if there are any intrinsics available that provide what you require. If no such intrinsics are available, use embedded assembler if absolutely necessary.

8.9.1 See also

Concepts

- *Restrictions on inline assembly operations in C and C++ code* on page 8-11
- *Compiler intrinsics* on page 5-3
- *Embedded assembler support in the compiler* on page 8-36.

Using the Assembler:

- *Processor modes, and privileged and unprivileged software execution* on page 3-5.

8.10 Inline assembler Thumb instruction set restrictions in C and C++ code

The inline assembler supports Thumb state in ARM architectures v6T2, v6M, and v7.

There are three other Thumb-specific restrictions:

1. `TBB`, `TBH`, `CBZ`, and `CBNZ` instructions are not supported.
2. In some cases, the compiler may replace `IT` blocks with branched code.
3. The instruction width specifier `.N` denotes a preference, but not a requirement, to the compiler. This is because, in rare cases, optimizations and register allocation can make it inefficient to generate a 16-bit encoding.

For ARMv6 and lower architectures, the inline assembler does not assemble any Thumb instructions. Instead, on finding inline assembly while in Thumb state, the compiler switches to ARM state automatically. Code that relies on this switch is currently supported, but this practise is deprecated. For ARMv6T2 and higher, the automatic switch from Thumb to ARM state will be made if the code is valid ARM assembly but not Thumb.

ARM state can be set deliberately. Inline assembly language can be included in a source file that contains code to be compiled for Thumb in ARMv6 and lower, by enclosing 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 for Thumb in ARMv6 and lower.

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 for Thumb in ARMv6 and
                // lower.
```

The code must also be compiled using the `--apcs /interwork` compiler command-line option.

8.10.1 See also

Concepts

- *Restrictions on inline assembly operations in C and C++ code* on page 8-11.

Reference

Assembler Reference:

- *CBZ and CBNZ* on page 3-50
- *TBB and TBH* on page 3-159
- *IT* on page 3-65
- *Instruction width specifiers* on page 3-8.

Compiler Reference:

- *--apcs=qualifer...qualifier* on page 3-11
- *Pragmas* on page 5-85.

8.11 Inline assembler *Vector Floating-Point* (VFP) restrictions in C and C++ code

The inline assembler provides direct support for VFPv2 instructions. For example:

```
float foo(float f, float g)
{
  float h;
    __asm
    {
        VADD h, f, 0.5*g; // h = f + 0.5*g
    }

    return h;
}
```

In inline assembler you cannot use the VFP instruction `VMOV` to transfer between an ARM register and half of a doubleword extension register (NEON scalar). Instead, you can use the instruction `VMOV` to transfer between an ARM register and a single-precision VFP register.

If you change the FPSCR register using inline assembler, it produces runtime effects on the inline VFP code and on subsequent compiler-generated VFP code.

Note

- Do not use inline assembly code to change VFP vector mode. Inline assembly code must not be used for this purpose, and VFP vector mode is deprecated.
- ARM strongly discourages the use of inline assembly coprocessor instructions to interact with VFP in any way.

8.11.1 See also

Concepts

- *Compiler support for floating-point arithmetic* on page 6-55
- *Restrictions on inline assembly operations in C and C++ code* on page 8-11.

Reference:

Assembler Reference:

- *VMOV (between an ARM register and a NEON scalar)* on page 4-64
- *VMOV (between one ARM register and single precision VFP)* on page 4-65.

8.12 Inline assembler instruction restrictions in C and C++ code

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.
- The special `LDM` instructions used in system or supervisor mode to load the user-mode banked registers, written with a `^` after the register list, such as:

  ```
  LDMIA sp!, {r0-r12, lr, pc}^
  ```

- `ADR` and `ADRL` pseudo-instructions.

  Note

  You can use `MOV Rn, &expression;` instead of the `ADR` and `ADRL` pseudo-instructions.

8.12.1 See also

Concepts

- *Restrictions on inline assembly operations in C and C++ code* on page 8-11.

Reference

- *__breakpoint intrinsic* on page 5-118.

8.13 Miscellaneous inline assembler restrictions in C and C++ code

Compared with `armasm` or embedded assembly language, 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 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 pointer. 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.

8.13.1 See also

Concepts

- *Restrictions on inline assembly operations in C and C++ code* on page 8-11.

8.14 Inline assembler and register access in C and C++ code

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 variable of the same name, and not the physical ARM register. (The variable can be thought of as a virtual register.)

The compiler declares variables for physical registers 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, or it might be stored on the stack. You can explicitly declare variables representing physical registers as normal C or C++ variables. The compiler implicitly declares registers `R0` to `R12` and `r0` to `r12` as **`auto signed int`** local variables, regardless of whether or not they are used. If you want to declare them to be of a different data type, you can do so. For example, in the following code, the compiler does not implicitly declare `r1` and `r2` as **`auto signed int`** because they are explicitly declared as **`char`** and **`float`** types respectively:

```
void bar(float *);

int add(int x)
{
  int a = 0;
  char r1 = 0;
  float r2 = 0.0;

    bar(&r2);
    __asm
    {
      ADD r1, a, #100
    }
    ...
    return r1;
}
```

The compiler does not implicitly declare variables for any other registers, so you must explicitly declare variables for registers other than `R0` to `R12` and `r0` to `r12` in your C or C++ code. No variables are declared for the `sp` (`r13`), `lr` (`r14`), and `pc` (`r15`) registers, and they cannot be read or directly modified in inline assembly code.

There is no virtual *Processor Status Register* (PSR). Any references to the `PSR` are always to the physical `PSR`.

The size of the variables is the same as the physical registers.

The compiler-declared variables have function local scope, that is, within a single function, multiple **`asm`** statements or declarations that reference the same variable name access the same virtual register.

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 variable representing a physical register is UNKNOWN. You must write to these variables before reading them. The compiler generates an error if you attempt to read such a variable before writing to it, for example, if you attempt to read the variable associated with the physical register `r1`.

Any variables that you use in inline assembler to refer to registers must be explicitly declared in your C or C++ code, unless they are implicitly declared by the compiler. However, it is better to *explicitly* declare them in your C or C++ code. You do not have to declare them to be of the

same data type as the implicit declarations. For example, although the compiler implicitly declares register `R0` to be of type **`signed int`**, you can explicitly declare `R0` as an unsigned integer variable if required.

It is also better to use C or C++ variables as instruction operands. The compiler generates a warning the first time a variable or physical register name is used, regardless of whether it is implicitly or explicitly declared, and only once for each translation unit. For example, if you use register `r3` without declaring it, a warning is displayed. You can suppress the warning with `--diag_suppress`.

8.14.1 See also

Concepts

- *Legacy inline assembler that accesses sp, lr, or pc* on page 8-56
- *Compiler support for inline assembly language* on page 8-4.

Reference

- *--diag_suppress=tag[,tag,...]* on page 3-73.

8.15 Inline assembler and the # constant expression specifier in C and C++ code

The constant expression specifier # is optional. If it is used, the expression following it must be a constant.

8.15.1 See also

Concepts

- *Compiler support for inline assembly language* on page 8-4.

8.16 Inline assembler and instruction expansion in C and C++ code

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.

See:

- *Expansion of inline assembler instructions that use constants* on page 8-22
- *Expansion of inline assembler load and store instructions* on page 8-23.

8.17 Expansion of inline assembler instructions that use 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
```

Another example of expansion possibility is:

```
MOV rn,0x12345678
```

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.

8.17.1 See also

Concepts

- *Inline assembler and instruction expansion in C and C++ code* on page 8-21
- *Compiler support for inline assembly language* on page 8-4.

8.18 Expansion of inline assembler 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. The warning can be suppressed with --diag_suppress.

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.

8.18.1 See also

Concepts

- *Inline assembler and instruction expansion in C and C++ code* on page 8-21
- *Compiler support for inline assembly language* on page 8-4.

Reference

Compiler Reference:

- *--diag_suppress=tag[,tag,...]* on page 3-73.

8.19 Inline assembler effect on processor condition flags in C and C++ code

An inline assembly language instruction might explicitly or implicitly attempt to update the processor condition flags. Inline assembly language instructions that involve only virtual register operands or simple expression 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.

8.19.1 See also

Concepts

- *Inline assembler expression operands in C and C++ code* on page 8-26
- *Inline assembler operands in C and C++ code* on page 8-25
- *Compiler support for inline assembly language* on page 8-4.

8.20 Inline assembler operands in C and C++ code

See the following topics for types of operands in inline assembler:

- *Inline assembler and register access in C and C++ code* on page 8-18
- *Inline assembler expression operands in C and C++ code* on page 8-26
- *Inline assembler register list operands in C and C++ code* on page 8-27
- *Inline assembler intermediate operands in C and C++ code* on page 8-28.

8.21 Inline assembler expression operands in C and C++ code

Function arguments, C or C++ variables, and other C or C++ expressions can be specified as register operands in an inline assembly language 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.

8.21.1 See also

Concepts

- *Inline assembler operands in C and C++ code* on page 8-25
- *Compiler support for inline assembly language* on page 8-4.

8.22 Inline assembler register list operands in C and C++ code

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 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.

8.22.1 See also

Concepts

- *Inline assembler operands in C and C++ code* on page 8-25
- *Compiler support for inline assembly language* on page 8-4.

8.23 Inline assembler intermediate operands in C and C++ code

A C or C++ constant expression of an integral type might be used as an immediate value in an inline assembly language 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.

8.23.1 See also

Concepts

- *Inline assembler operands in C and C++ code* on page 8-25
- *Compiler support for inline assembly language* on page 8-4.

Other information

- *ARM Architecture Reference Manual*.

8.24 Inline assembler function calls and branches in C and C++ code

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.

If you are compiling for architecture 5TE or later, the linker converts BL *function* instructions to BLX *function* instructions if appropriate. However, you cannot use BLX *function* instructions directly within inline assembly code.

The lists are described in the following topics:

- *Behavior of BL and SVC without optional lists in C and C++ code* on page 8-30
- *Inline assembler BL and SVC input parameter list in C and C++ code* on page 8-31
- *Inline assembler BL and SVC output value list in C and C++ code* on page 8-32
- *Inline assembler BL and SVC corrupted register list* on page 8-33.

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.

8.25 Behavior of BL and SVC without optional lists in C and C++ code

The BL and SVC instructions of the inline assembler 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}
```

If you do not specify any lists, then:

- r0-r3 are used as input parameters
- r0 is used for the output value and can be corrupted
- r0-r3, r14, and the PSR can be corrupted.

8.25.1 See also

Concepts

- *Inline assembler function calls and branches in C and C++ code* on page 8-29
- *Compiler support for inline assembly language* on page 8-4.

8.26 Inline assembler BL and SVC input parameter list in C and C++ code

The BL and SVC instructions of the inline assembler 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}
```

input_param_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 assembly language 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
```

By default, if you do not specify any *input_param_list* input parameters, registers r0 to r3 are used as input parameters.

Note

It is not possible to specify the lr, sp, or pc registers in the input parameter list.

8.26.1 See also

Concepts

- *Inline assembler function calls and branches in C and C++ code* on page 8-29
- *Compiler support for inline assembly language* on page 8-4.

8.27 Inline assembler BL and SVC output value list in C and C++ code

The BL and SVC instructions of the inline assembler 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}
```

output_value_list specifies the physical registers that contain the output values from the BL or SVC instruction 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
```

By default, if you do not specify any *output_value_list* output values, register r0 is used for the output value.

Note

It is not possible to specify the lr, sp, or pc registers in the output value list.

8.27.1 See also

Concepts

- *Inline assembler function calls and branches in C and C++ code* on page 8-29
- *Compiler support for inline assembly language* on page 8-4.

8.28 Inline assembler BL and SVC corrupted register list

The BL and SVC instructions of the inline assembler 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}
```

corrupt_reg_list specifies the physical registers that are corrupted by the called function. If the condition flags are modified by the called function, you must specify the PSR in the corrupted register list.

The BL and SVC instructions always corrupt lr.

If *corrupt_reg_list* is omitted then for BL and SVC, the registers r0-r3, lr and the PSR are corrupted.

Only the branch instruction, B, can be used to jump to labels within a single C or C++ function.

By default, if you do not specify any *corrupt_reg_list* registers, r0 to r3, r14, and the PSR can be corrupted.

Note

It is not possible to specify the lr, sp, or pc registers in the corrupt register list.

8.28.1 See also

Concepts

- *Inline assembler function calls and branches in C and C++ code* on page 8-29
- *Compiler support for inline assembly language* on page 8-4.

8.29 Inline assembler branches and labels in C and C++ code

Labels defined in inline assembly code:

- can be used as targets for branches or C and C++ `goto` statements
- must be followed by a colon, `:`, like C and C++ labels
- must be within the same function that they are called from.

Labels defined in C and C++ can be used as targets by branch instructions in inline assembly code, in the form:

```
B{cond} label
```

For example:

```
int foo(int x, int y)
{
  __asm
  {
     SUBS x,x,y
     BEQ end
  }

  return 1;

  end:
    return 0;
}
```

8.29.1 See also

Concepts

- *Compiler support for inline assembly language* on page 8-4.

8.30 Inline assembler and 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 fully optimize 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;
}
```

8.30.1 See also

Concepts

- *Inline assembler and register access in C and C++ code* on page 8-18.

8.31 Embedded assembler support in the compiler

The 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. The embedded assembler supports ARM and Thumb states.

8.31.1 See also

Concepts

- *Embedded assembler syntax in C and C++* on page 8-37
- *Effect of compiler ARM and Thumb states on embedded assembler* on page 8-39
- *Restrictions on embedded assembly language functions in C and C++ code* on page 8-40
- *Differences between expressions in embedded assembler and C or C++* on page 8-44
- *Compiler generation of embedded assembly language functions* on page 8-41
- *Access to C and C++ compile-time constant expressions from embedded assembler* on page 8-43
- *Manual overload resolution in embedded assembler* on page 8-45
- *__offsetof_base keyword for related base classes in embedded assembler* on page 8-46
- *Compiler-supported keywords for calling class member functions in embedded assembler* on page 8-47
- *Calling nonstatic member functions in embedded assembler* on page 8-53
- *Calling a nonvirtual member function* on page 8-54
- *Calling a virtual member function* on page 8-55.

Developing Software for ARM Processors:

- Chapter 4 *Mixing C, C++, and Assembly Language*.

Assembler Reference:

- Chapter 2 *Assembler command line options*.

8.32 Embedded assembler syntax in C and C++

An embedded assembly language function definition is marked by the **__asm** function qualifier in C and C++, or the **asm** function qualifier in C++, 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 language function is:

```
__asm return-type function-name(parameter-list)
{
   // ARM/16-bit Thumb/32-bit Thumb assembler code
   instruction{;comment is optional}
   ...
   instruction
}
```

Note

Argument names are permitted in the parameter list, but they cannot be used in the body of the embedded assembly function. For example, the following function uses integer `i` in the body of the function, but this is not valid in assembly:

```
__asm int f(int i)
{
    ADD i, i, #1 // error
}
```

You can use, for example, `r0` instead of `i`.

Example 8-1 shows a string copy routine as a not very optimal embedded assembler routine.

Example 8-1 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;
}
```

8.32.1 See also

Concepts

- *Embedded assembler support in the compiler* on page 8-36.

8.33 Effect of compiler ARM and Thumb states on embedded assembler

The initial state of the embedded assembler, ARM or Thumb state, 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 32-bit Thumb capable processor, you can use 32-bit Thumb instructions when in Thumb state.

8.33.1 See also

Concepts

- *Embedded assembler support in the compiler* on page 8-36.

8.34 Restrictions on embedded assembly language functions in C and C++ code

The following restrictions apply to embedded assembly language functions:

- After preprocessing, `__asm` functions can only contain assembly code, with the exception of the following embedded assembler built-ins:

  ```
  __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, 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 define them. However, inlined and template functions behave differently. Do not assume that code execution falls out of an inline or template function into another embedded assembly function.

- `__asm` functions do not change the *ARM Architecture Procedure Call Standard* (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).

8.34.1 See also

Concepts

- *__offsetof_base keyword for related base classes in embedded assembler* on page 8-46
- *Compiler-supported keywords for calling class member functions in embedded assembler* on page 8-47
- *Compiler generation of embedded assembly language functions* on page 8-41
- *Embedded assembler support in the compiler* on page 8-36
- *Access to C and C++ compile-time constant expressions from embedded assembler* on page 8-43.

8.35 Compiler generation of embedded assembly language 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 language 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 8-2:

Example 8-2 `__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 `__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 are 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, do not assume that code execution falls out of an inline or template function and into another `__asm` function.

8.35.1 See also

Concepts

- *Embedded assembler support in the compiler* on page 8-36.

Other information

- *ARM ELF File Format*, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0101-

8.36 Access to C and C++ compile-time constant expressions from embedded assembler

You can use the **__cpp** keyword to access C and 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 8-3 shows a constant replacing the use of `__cpp(expr)`:

Example 8-3 `__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.

8.36.1 See also

Concepts

- *Embedded assembler support in the compiler* on page 8-36
- *Manual overload resolution in embedded assembler* on page 8-45
- *Differences between expressions in embedded assembler and C or C++* on page 8-44.

Other information

- *ISO/IEC 14882:2003*
  - Section 3.6.2 Initialization of non-local objects
  - Section 5.19 Constant expressions.

8.37 Differences between expressions in embedded assembler and C or C++

The following differences exist 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 NUL-terminated:

```
DCB "Hello world!"                // 12 bytes (no trailing NUL) DCB __cpp("Hello world!")         // 13 bytes (trailing NUL)
```

Note

The assembler rules apply outside `__cpp`, and the C or C++ rules apply inside `__cpp`.

8.37.1 See also

Concepts

- *Access to C and C++ compile-time constant expressions from embedded assembler* on page 8-43
- *Embedded assembler support in the compiler* on page 8-36.

8.38 Manual overload resolution in embedded assembler

Example 8-4 shows the use of C++ casts to do overload resolution for nonvirtual function calls:

Example 8-4 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
}
```

8.38.1 See also

Concepts

- *Embedded assembler support in the compiler* on page 8-36.

8.39 __offsetof_base keyword for related base classes in embedded assembler

The __offsetof_base 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, nonvirtual 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 8-5 shows the offset (in bytes) that must be added to a D* p to implement the equivalent of static_cast<B*>(p).

Example 8-5 static_cast<B*>(p)

```
__asm B* my_static_base_cast(D* /*p*/) // equivalent to:
                                       // return static_cast<B*>(p)
{
    if __offsetof_base(D, B) <> 0 // optimize zero offset case
        ADD r0, r0, #__offsetof_base(D, B)
    endif
    BX lr
}
```

The **__offsetof_base**, **__mcall_***, and **_vcall_offsetof_vfunc** keywords are converted into integer or logical constants in the assembler source. You can only use it in __asm functions, not in __cpp expressions.

8.39.1 See also

Concepts

- *Restrictions on embedded assembly language functions in C and C++ code* on page 8-40
- *Embedded assembler support in the compiler* on page 8-36.

8.40 Compiler-supported keywords for calling class member functions in embedded assembler

The following embedded assembler built-ins facilitate the calling of virtual and nonvirtual member functions from an `__asm` function. Those beginning with `__mcall` can be used for both virtual and nonvirtual functions. Those beginning with `__vcall` can be used only with virtual functions. They do not particularly help in calling static member functions.

8.41 __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.

8.41.1 See also

Concepts

- *Compiler-supported keywords for calling class member functions in embedded assembler* on page 8-47.

8.42 __mcall_is_in_vbase(D, f)

Results in {TRUE} if f is a nonstatic 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).

8.42.1 See also

Concepts

- *Compiler-supported keywords for calling class member functions in embedded assembler* on page 8-47
- *Calling a nonvirtual member function* on page 8-54
- *Calling a virtual member function* on page 8-55.

8.43 __mcall_offsetof_vbase(D, f)

Where D is a class type and f is a nonstatic member function defined in a virtual base class of D, in other words __mcall_is_in_vbase(D,f) returns {TRUE}.

__mcall_offsetof_vbase() returns the negative offset from the value of the vtable pointer of the vtable slot that holds the base offset (from the beginning of a D object to the start of the base that f is defined in).

The base offset 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 value of the vtable pointer.

8.43.1 See also

Tasks

- *Calling a nonvirtual member function* on page 8-54
- *Calling a virtual member function* on page 8-55.

Concepts

- *Compiler-supported keywords for calling class member functions in embedded assembler* on page 8-47.

8.44 __mcall_this_offset(D, f)

Where D is a class type and f is a nonstatic member function defined in D or a nonvirtual 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 nonvirtual 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.

8.44.1 See also

Tasks

- *Calling a nonvirtual member function* on page 8-54
- *Calling a virtual member function* on page 8-55.

Concepts

- *Compiler-supported keywords for calling class member functions in embedded assembler* on page 8-47.

8.45 __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 offset of the slot in the vtable that holds the pointer to the virtual function, `f`.

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.

8.45.1 See also

Tasks

- *Calling a virtual member function* on page 8-55.

Concepts

- *Compiler-supported keywords for calling class member functions in embedded assembler* on page 8-47.

8.46 Calling nonstatic member functions in embedded assembler

You can use keywords beginning with `__mcall` and `__vcall` to call nonvirtual and virtual functions from `__asm` functions. There is no `__mcall_is_static` to detect static member functions because static member functions have different parameters (that is, no `this`), so call sites are likely to already be specific to calling a static member function.

8.46.1 See also

Concepts

- *Compiler-supported keywords for calling class member functions in embedded assembler* on page 8-47
- *Calling a nonvirtual member function* on page 8-54
- *Calling a virtual member function* on page 8-55.

8.47 Calling a nonvirtual member function

Example 8-6 shows code that can be used to call a nonvirtual function in either a virtual or nonvirtual base:

Example 8-6 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 set up
// 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
```

8.47.1 See also

Concepts

- *Calling nonstatic member functions in embedded assembler* on page 8-53.

8.48 Calling a virtual member function

Example 8-7 shows code that can be used to call a virtual function in either a virtual or nonvirtual base:

Example 8-7 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()
```

8.48.1 See also

Concepts

- *Calling nonstatic member functions in embedded assembler* on page 8-53.

8.49 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. Example 8-8 shows how legacy inline assembly code might use `lr`.

Example 8-8 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
- 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.

8.49.1 See also

Tasks

- *Accessing sp (r13), lr (r14), and pc (r15) in legacy code* on page 8-57.

Concepts

- *Compiler support for inline assembly language* on page 8-4
- *Embedded assembler support in the compiler* on page 8-36.

Reference

- *Instruction intrinsics* on page 5-117.

8.50 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.

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.

8.50.1 See also

Concepts

- *Embedded assembler support in the compiler* on page 8-36.

Reference

- *__return_address intrinsic* on page 5-150

Compiler Reference:

- *__current_pc intrinsic* on page 5-122
- *__current_sp intrinsic* on page 5-123.

8.51 Differences in compiler support of inline and embedded assembly code

There are differences between the ways inline and embedded assembly are 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 8-1 summarizes the main differences between inline assembler and embedded assembler.

Table 8-1 Differences between inline and embedded assembler

Feature	Embedded assembler	Inline assembler
Instruction set	ARM and Thumb.	ARM on all processors. Thumb on processors with Thumb-2 technology.
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.	Supports most instructions.
VFP and NEON instructions	All supported.	VFPv2 only.
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. Using `sp` (`r13`), `lr` (`r14`), and `pc` (`r15`) gives an error.
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.

8.51.1 See also

Concepts

- *Differences between expressions in embedded assembler and C or C++* on page 8-44
- *Inline assembler and register access in C and C++ code* on page 8-18
- *Legacy inline assembler that accesses sp, lr, or pc* on page 8-56
- *Compiler support for inline assembly language* on page 8-4
- *Embedded assembler support in the compiler* on page 8-36
- *Inline assembler support in the compiler* on page 8-5.

Appendix A
Revisions for Using the Compiler

This appendix describes the technical changes between released issues of this book.

Table A-1 Differences between issue G and issue H

Change	Topics affected
Added information on estimating/measuring stack usage.	*Stack use in C and C++* on page 6-17.
Clarified which `LDM` instructions are not supported in the inline assembler.	*Inline assembler instruction restrictions in C and C++ code* on page 8-16.
Documented Cortex-M0+ support.	*Processors and their implicit Floating-Point Units (FPUs)* on page 6-71.
Added two topics.	• *Floating-point linkage and computational requirements of compiler options* on page 6-69 • *Processors and their implicit Floating-Point Units (FPUs)* on page 6-71.

Table A-2 Differences between issue F and issue G

Change	Topics affected
Clarified the difference between `__packed` and `#pragma` pack for address-taken fields.	*Comparisons of an unpacked struct, a __packed struct, and a struct with individually __packed fields, and of a __packed struct and a #pragma packed struct* on page 6-52.
Where appropriate, rather than 16-bit Thumb or 32-bit Thumb, referred instead to Thumb or Thumb-2 technology.	Various topics.
Added three entries for VFPv4.	*Vector Floating-Point (VFP) architectures* on page 6-60.
Added Cortex-A7.	• *Generating NEON instructions from C or C++ code* on page 4-7 • *Vectorizable code example* on page 4-34 • *DSP vectorizable code example* on page 4-37 • *What can limit or prevent automatic vectorization* on page 4-40.
Revised the topics related to integer division by zero.	• *Integer division-by-zero errors in C code* on page 6-75 • *About trapping integer division-by-zero errors with __aeabi_idiv0()* on page 6-76 • *About trapping integer division-by-zero errors with __rt_raise()* on page 6-77.

Table A-3 Differences between issue D and issue F

Change	Topics affected
Revised the descriptions for topics related to integer division by zero.	• *Integer division-by-zero errors in C code* on page 6-75 • *About trapping integer division-by-zero errors with __aeabi_idiv0()* on page 6-76 • *About trapping integer division-by-zero errors with __rt_raise()* on page 6-77.
Mentioned that VFPv2 instructions are supported in inline assembly. Mentioned that instructions in VFPv3 or higher are not supported by inline assembler.	• *Inline assembler support in the compiler* on page 8-5 • *Restrictions on inline assembler support in the compiler* on page 8-6 • *Inline assembler Vector Floating-Point (VFP) restrictions in C and C++ code* on page 8-15.
Changed the description of the restrictions on the inline assembler.	• *Restrictions on inline assembler support in the compiler* on page 8-6 • *Inline assembler Thumb instruction set restrictions in C and C++ code* on page 8-14 • *Differences in compiler support of inline and embedded assembly code* on page 8-58.

Table A-3 Differences between issue D and issue F (continued)

Change	Topics affected
Removed the example of using coprocessor instructions in inline assembler because coprocessor instructions in inline assembler are discouraged. Added a new example for the use of VFP instructions in inline assembler.	*Inline assembler Vector Floating-Point (VFP) restrictions in C and C++ code* on page 8-15.
Added a row on VFP and NEON for differences between inline and embedded assembler.	*Differences in compiler support of inline and embedded assembly code* on page 8-58.
Where appropriate: • prefixed Thumb with 16-bit • changed Thumb-1 to 16-bit Thumb • changed Thumb-2 to 32-bit Thumb.	Various topics.
Changed the ARMCC*nn*_CCOPT and ARMCC*nn*INC environment variables to ARMCC*n*_CCOPT and ARMCC*n*INC.	Various topics.

Table A-4 Differences between issue C and issue D

Change	Topics affected
Added a topic on using GCC fallback.	*Using GCC fallback when building applications* on page 3-24.
Removed the topics on Profiler-guided optimizations.	Chapter 5 *Compiler Features*.
Removed reference to ARM Profiler.	*Code metrics* on page 6-15.
Removed references to Eclipse Workbench IDE and ARM Profiler.	*Code metrics for measurement of code size and data size* on page 6-16.
Merged the following topics into the topic Stack use in C and C++: • Methods of estimating stack usage • Methods of reducing stack usage • Using a debugger to estimate stack usage. Changed `--info=summary` stack to `--info=summarystack`. Also removed references to ARM Profiler. Rephrased branches to function calls.	*Stack use in C and C++* on page 6-17.
Removed the note about profiler guided optimizations	*Inline functions* on page 6-30.
Removed reference to `--profile`	*Inline functions and removal of unused out-of-line functions at link time* on page 6-34.
Changed `ARMCC41*` environment variables to `ARMCCnn*`. And added link to the topic Toolchain environment variables in the *Introducing to ARM Compiler toolchain* document.	Various topics.

Table A-4 Differences between issue C and issue D (continued)

Change	Topics affected
Changed ARM Compiler 4.1 to ARM Compiler 4.1 and later.	• *Automatic function inlining and multifile compilation* on page 6-35 • *Compiler eight-byte alignment features* on page 5-31 • *Compiler support for European Telecommunications Standards Institute (ETSI) basic operations* on page 5-13 • *Generating NEON instructions from C or C++ code* on page 4-7 • *Effect of --multifile on compilation build time* on page 3-34 • *How to minimize compilation build time* on page 3-31.
Removed mention of Vista.	*PreCompiled Header (PCH) files* on page 5-33.
Removed mention of Solaris.	• *How to minimize compilation build time* on page 3-31 • *Minimizing compilation build time with parallel make* on page 3-35 • *Compilation build time and operating system choice* on page 3-36.
Changed onwards to later, and mentioned ARM Compiler 4.1.	• *The NEON unit* on page 4-4 • *Compiler intrinsics for Digital Signal Processing (DSP)* on page 5-11.
Removed mention of FLEXlm licence.	*Compilation build time* on page 3-29.
When -J is not specified on the command-line, mention that the compiler searches `ARMCC`*`nn`*`INC`, then `ARMINC`, then `../include`.	*Compiler command-line options and search paths* on page 3-19.
Added `ARMINC` to the list of search paths.	*Factors influencing how the compiler searches for header files* on page 3-18.
Added link to Compiler command line options and search paths.	*The ARMCCnINC environment variable* on page 3-21.
Changed description to say that `ARMCC`*`nn`*`INC` might be initialized.	*The ARMCCnINC environment variable* on page 3-21.
Moved "…is colon separated on UNIX …" to the topic on `ARMCC`*`nn`*`INC`.	*The ARMCCnINC environment variable* on page 3-21.
Specify the search order when `-I` and `-J` are both specified.	*Compiler command-line options and search paths* on page 3-19.
Added note to deprecate `--ltcg`.	*Automatic function inlining and multifile compilation* on page 6-35.
Inline assembler definitions with **`__asm`** and **`asm`** can include multiple strings.	• *Inline assembly language syntax with the __asm keyword in C and C++* on page 8-7 • *Inline assembly language syntax with the asm keyword in C++* on page 8-8 • *Inline assembler rules for compiler keywords __asm and asm* on page 8-9.

Table A-5 Differences between issue B and issue C

Change	Topics affected
Changed the doubleword alignment to be multiples of 8 instead of 4.	*Advantages of natural data alignment* on page 6-43.

Table A-6 Differences between issue A and issue B

Change	Topics affected
New topic.	*Default compiler options that are affected by optimization level* on page 5-45.
New topic.	*How the compiler handles bit-band objects placed outside bit-band regions* on page 5-29.
Filename suffixes `.i`, `.ii`, `.a`, `.lib`, `.so` added.	*Filename suffixes recognized by the compiler* on page 3-15.
Usage notes for `.S` suffix.	*Filename suffixes recognized by the compiler* on page 3-15.
Description for `.sx` suffix.	*Filename suffixes recognized by the compiler* on page 3-15.
Search criteria for `armlink` and `armasm` executables.	*Default compiler behavior* on page 3-9.
Example code for `promise.c`	*Indicating loop iteration counts to the compiler with __promise(expr)* on page 4-25.
New topic.	*How to prevent uninitialized data from being initialized to zero* on page 6-109.